ediciones técnicas



REDE

BARCELONA
(ESPAÑA)

J. SALES ROIG

programación en CODIGO MAQUINA para el ZX81 y para el Spectrum

REDE

programación en CODIGO MAQUINA para el ZX81 y para el Spectrum

por

JOAN SALES ROIG

Guía práctica para programar en CM



REDE

JOAN SALES ROIG

PROGRAMACION EN CODIGO MAQUINA PARA EL ZX81 Y PARA EL SPECTRUM

EDICIONES TECNICAS REDE, S.A.
Apartado 35.400
BARCELONA

ISBN 84-247-0193-3

Impreso en España
Printed in Spain
D.L.: B. 11284-1984
—REDEPRINT—
Barcelona

PROLOGO

No hay duda de que los micro-ordenadores SINCLAIR ZX81 y ZX-SPECTRUM han puesto la informática al alcance de todos, de modo que ahora es posible disponer de un ordenador en casa para llevar la gestión casera y profesional, e incluso como ayuda en el trabajo. La mayoría de estas tareas pueden ser fácilmente programadas en BASIC con excelentes resultados y sin mayores dificultades, ya que con la ayuda del manual que se suministra con cada ordenador es posible aprender a programarlo efectivamente en BASIC.

Sin embargo, muchos de los usuarios de los ZX están más interesados en programar juegos en los que se requiere gráficos animados moviéndose rápidamente en la pantalla, así como respuestas instantáneas a los controles del teclado. En este aspecto, programando el ZX en BASIC se obtienen resultados bastante satisfactorios. Ello es debido a que para poder suministrar estos micro-ordenadores a precios al alcance de la mayoría de los bolsillos, ha habido que sacrificar algunos detalles técnicos que normalmente tienen los ordenadores de mucho mayor precio y sustituirlos por mecanismos que influyen decisivamente en la velocidad operativa de todo el conjunto.

Estos inconvenientes pueden ser sencillamente superados programando los ZX en el lenguaje nativo de su microprocesador, el Z80, esto es, programándolo en código máquina. De este modo, al evitar el paso intermedio que invariablemente realiza la máquina de traducir cada línea de programa BASIC a código máquina, se gana extraordinariamente en velocidad, lo que hace posible programar los juegos más complicados.

Este es el primer libro original publicado en español sobre programación de los ZX en código máquina y, puesto que no hay ninguna explicación sobre este tema en los manuales que acompañan a los ordenadores, resulta imprescindible para todos los interesados en aprender a programar en código máquina y dominar y conocer los secretos más recónditos de los ZX.

Dado que su autor, Joan Sales Roig, ha aprendido a programar en código máquina sin ningún conocimiento previo sobre este tema, esta obra ha sido escrita de modo que introduce progresivamente los conceptos más elementales hasta desembocar en los más complejos, de la misma manera en que lo ha aprendido el propio autor, por lo que, mediante una lectura y desarrollo progresivo y ordenado del mismo, es posible llegar a dominar la programación en código máquina de los ZX.

Actualmente, Joan Sales se dedica a la programación de juegos en el ZX-SPECTRUM, que están teniendo un gran éxito en la misma cuna de los ZX, en Inglaterra.

Josep-Oriol Tomàs
Coordinador del Club Nacional de Usuarios de los ZX

INTRODUCCION

Por qué un libro sobre CM (Código Máquina) para el ZX81

Es un hecho plenamente demostrado que para conseguir un rendimiento máximo del ZX81 hay que programarlo en código máquina. La diferencia de resultados entre un programa en CM y otro en BASIC es mayor en el ZX81 que en otros micros porque al tener un precio muy bajo la mayor parte del trabajo se hace por software, con lo que al trabajar en BASIC la velocidad de trabajo no es muy elevada. Además, y como consecuencia de esto, hay una serie de procesos «principales» que no conviene interferir, y que hay que conocer al realizar un programa en código máquina.

Se han publicado varios libros en el extranjero sobre este tema, pero hasta ahora no se había hecho nada en España. Este libro no es una traducción ni una adaptación de los que ya existen, sino que está pensado y escrito en español y todo el material, rutinas y programas que contiene son originales y pensados especialmente para dar mayor claridad a los conceptos «teóricos».

El planteamiento del libro se ha aproximado todo lo posible al de un cursillo, donde cada tema se trata desde cero y se va ampliando a medida que avanza el capítulo. En cada capítulo se da por entendido todo lo que le precede, por lo que es muy recomendable empezar por el principio y seguir los capítulos por orden, sin prisas.

Material necesario y nivel de partida

El libro está pensado para personas que conozcan el BASIC y su funcionamiento, aunque no lo dominen, pero que no tienen ni idea sobre código máquina ni sobre microprocesadores. Cada capítulo trata una serie de conceptos relacionados, partiendo desde cero hasta

niveles más complejos. Según los conocimientos iniciales cada lector encontrará en cada capítulo su punto de partida. Conviene también tener en cuenta que a veces se tienen conceptos erróneos de los que se está muy seguro, y nunca viene mal refrescar un poco la memoria.

Aunque siempre es recomendable disponer de una ampliación de memoria RAM para sacar un buen partido del ZX81, todos los programas y ejercicios pueden realizarse con el equipo básico: el ZX81 con 1K de memoria, un TV y un cassette. No se necesita ningún tipo de ampliación. Se puede trabajar con más memoria si se toman las precauciones que se indican en los lugares adecuados. Dado que la organización de la memoria de pantalla depende de la cantidad de RAM disponible, se tratarán con detalle y por separado los diferentes casos.

Plan general del libro

El material de este libro está organizado en 20 capítulos, agrupados en tres secciones:

1. Introducción y conceptos generales de informática y programación de microprocesadores, aunque orientados hacia el microprocesador Z80 (el del Sinclair ZX81). Comprende los capítulos 1, 2, 3.
2. Instrucciones del lenguaje CM del Z80 y la forma de empleo al programar con el ZX81. Es una visión muy detallada que ocupa los capítulos 5 al 13 inclusive.
3. Estudio detallado de la programación del teclado y las salidas MIC y EAR del ZX81, técnicas para realizar buenos programas, su aplicación a un programa complejo (el «COMECOCOS» visto con todo detalle) y direcciones útiles de subrutinas de la ROM que pueden aprovecharse.

No se ha pretendido hacer de este libro un conjunto de recetas de programación ni un paquete de programas «espectaculares» para verlos funcionar sino una herramienta de trabajo para que cada uno pueda aprender a programar el ZX81 en código máquina y, lo que es más importante, a programarlo bien.

CAPÍTULO PRIMERO

DECIMAL, HEXADECIMAL, BINARIO

1.1. Sistemas de numeración: BIT, BYTE, KBYTE

Al menos por el momento, se puede considerar que un ordenador no es más que una máquina de calcular pero extraordinariamente perfeccionada. Lo que puede hacer en un momento dado es extremadamente sencillo, pero puede hacerlo muy de prisa. Es como tocar el piano: no hay más que apretar las teclas, pero un pasaje que un concertista ejecuta en un minuto, a alguien que no sepa tocar el piano le llevará toda una tarde pulsar las mismas teclas. Además, con sólo 64 teclas se pueden interpretar infinitas piezas.

En nuestro caso las «teclas» del ordenador son los números. El problema está en que el ordenador no cuenta como el hombre. Nosotros empleamos para contar un sistema que tiene diez dígitos: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. Cuando hemos agotado los diez dígitos volvemos a empezar con el cero pero aumentando en una unidad el contador que indica el número de veces que hemos completado la serie: este contador será la cifra de las decenas. Por ejemplo, 37 significa que hemos contado tres veces hasta diez y luego hasta siete.

La única razón por la que hacemos este salto al llegar a diez y no antes o después es porque tenemos diez dedos en las manos y la primera «máquina de calcular» que se inventó fueron los dedos. Este sistema de numeración se llama *DECIMAL* o de *base diez*.

Pero supongamos ahora que en lugar de diez dedos tuviéramos dieciséis: manejaríamos dieciséis dígitos, que podemos llamar 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, y F. Con este sistema, al llegar a 9 no tenemos por qué volver a empezar, pues aún nos quedan dígitos, y por tanto después de 9 viene A, luego B, C, D, E y F; entonces sí

saltamos y tras F viene 10, 11, 12,... 19, 1A, 1B, 1C,... 1F, 20, 21, 22,... 9E, 9F, A0, A1, A2,... A9, AA, AB,... AF, B0, B1,... etc. Este sistema que tiene dieciséis dígitos se llama *HEXADECIMAL* o de *base dieciséis.*

Si no tuviéramos más que dos dedos (un ordenador «sólo tiene dos dedos»), únicamente tendríamos dos dígitos, 0 y 1 y la numeración sería: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011,... Este sistema con dos dígitos se llama sistema *BINARIO* o de *BASE DOS.* En la tabla 1 puede verse la equivalencia entre los tres sistemas de numeración hasta 255.

TABLA 1. CONVERSION HEXADECIMAL

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

RUTINA DISPLAY BIN–HEX–DEC

Parte en BASIC:

```
1 REM ... código máquina
10 PRINT "00000000!BIN!$00!!!!"; (! = espacio)
20 IF INKEY$ <> "" THEN GOTO 20
30 IF INKEY$ = "" THEN GOTO 30
40 PRINT AT 0,20; USR 16514
50 GOTO 20
```


Para poner el contador a cero: "POKE 16507,0"

Parte en código máquina:

```
INICIO     LD HL,16507
           LD A,(16922)
           CP 247
           JR Z, INC
           CP 239
           JR Z, DEC
           CP 253
           JR Z, CERO
           LD A,(16507)
           LD C,A
           LD B,0
           RET
CERO       LD (HL),0
           JR ACTUALIZA
INC        INC (HL)
           JR ACTUALIZA
DEC        DEC (HL)
           ACTUALIZA LD A,(HL)
           LD HL,(16396)
           INC HL
           LD B,8
DIGIT      RLCA
           JR NC ES CERO
           LD (HL),29
           JR CONT
ES CERO    LD (HL),28
CONT       INC HL
           DJNZ DIGIT
           LD DE,7
           ADD HL,DE
           LD A,(16507)
           PUSH AF
           AND $F0
           RRA
           RRA
           RRA
           RRA
           ADD A,28
```

```
LD (HL), A
INC HL
POP AF
AND $0F
ADD A,28
LD (HL),A
RET
```

Pulsando la tecla «I» se incrementa el contador

«R» se decrementa

«0» se pone a cero

Secuencia de códigos Hex de la parte en CM de la rutina, dispuestos en el mismo orden en que deben ser entrados con el CARGADOR HEXADECIMAL:

21	7B	40	3A	1A	42	FE	F7
28	13	FE	EF	28	12	FE	FD
28	07	3A	7B	40	4F	06	00
C9	36	00	18	04	34	18	01
35	7E	2A	Oc	40	23	06	08
07	30	04	36	1D	18	02	36
1C	23	10	F4	11	07	00	19
3A	7B	40	F5	E6	F0	1F	1F
1F	1F	C6	1C	77	23	F1	E6
0F	C6	1C	77	C9			

La línea «1 REM» deberá tener 78 caracteres.

La menor cantidad de información posible es, pues, un «1» o un «0», que en el ordenador se almacena físicamente como el paso o no de corriente por un punto determinado. Este «lugar», que puede contener tan sólo 1 ó 0, se llama un *BIT* de información. Una gran parte del ordenador está formada por bits que soportan cada uno una pequeña parte de la información total, sea ésta un programa, datos, o cualquier otra cosa.

Los bits se agrupan para su gestión en unidades mayores: Cada ocho bits consecutivos forman un *BYTE*, y son tratados en bloque generalmente por el ordenador. Con ocho bits podemos formar números entre cero y 255, o entre $00 y $SFF Hex. (Tanto el símbolo «$»

como «Hex.» se utilizarán en este libro para indicar que un número está expresado en hexadecimal). Los bits se agrupan de 8 en 8 por razones técnicas. De hecho los primeros microprocesadores eran de 4 bits y actualmente empiezan a proliferar los de 16 bits, aunque el ZX81 trabaja con uno de 8 bits, y un byte son siempre 8 bits, sea cual sea el microprocesador que lo gestione.

Cada 1024 bytes se agrupan en una unidad superior: un kilobyte o Kbyte, representado generalmente por una «K». Según esto, 16K de memoria son 16 × 1024 = 16.384 bytes o 16.384 × 8 = = 111.072 bits o posiciones que pueden contener en uno o un cero.

Cada byte se identifica por un número de orden dentro de la memoria, llamado dirección o posición de memoria. Es como una serie de «cajas» numeradas de 0 en adelante que pueden contener un número entre 0 y 255 cada una. En CM trabajamos directamente con las direcciones de memoria, de modo que escribimos un dato en la dirección de memoria requerida, en lugar de trabajar con nombres de variables.

Estas posiciones de memoria están agrupadas en bloques lógicos que estructuran la memoria total del ZX81. La disposición de estos bloques se analiza en el capítulo 2.

Para comprender mejor las analogías y diferencias entre los sistemas de numeración hexadecimal, decimal y binario puede ser útil experimentar un poco con la siguiente rutina, que va presentando en pantalla el mismo número en los tres sistemas, incrementándose o decrementándose en uno cada vez que se pulsa una tecla determinada.

Los bits de un byte se numeran de izquierda a derecha: 7, 6, 5, 4, 3, 2, 1 y 0. Para pasar un byte expresado en binario a decimal se puede aplicar la siguiente fórmula:

b7b6b5b4b3b2b1b0 es el byte en binario: b7 es el bit 7, b6 el bit 6, etc.

b7b6b5b4b3b2b1b0 binario = 128 × b7 + 64 × b6 + 32 × b5 + + 16 × b4 + 8 × b3 + 4 × b2 + 2 × b1 + b0.

Por ejemplo: 10110101 = 128 × 1 + 64 × 0 + 32 × 1 + 16 × 1 + + 8 × 0 + 4 × 1 + 2 × 0 + 1 = 128 + 32 + 16 + 4 + 1 = = 181 decimal.

Con este sistema sólo podemos tener números positivos. Hay ocasiones en las que necesitamos que un número pueda ser positivo o negativo, incluso algunas instrucciones en CM requieren números con signo. Para dar signo a un número se recurre a utilizar un bit del propio número para indicarlo, concretamente el bit 7. Si el bit 7 es

cero, el número es positivo y si es uno es negativo. Al utilizar uno de los bits para indicar el signo, el número ya no podrá estar entre 0 y 255 sino entre +127 y −128 (127 + 128 más el cero son los 256 números diferentes que podemos representar con 8 bits).

Para sumar dos números en binario hay que sumar bit a bit según la siguiente regla:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 y sumamos 1 al siguiente bit («llevamos una»).

La operación es por lo demás como en decimal, empezando por la derecha.

El problema está en que el sistema para dar signo a los números que hemos visto no funciona. Si sumamos +7 y −5:

(+7) 00000111
(−5) 10000101
10001100 que puede comprobarse que es −12.

Para hallar la solución a esto necesitamos trabajar con el complementario.

El complementario de un número binario se obtiene cambiando simplemente cada uno por un cero y viceversa. Así el complementario de 10010110 será 01101001. Según esto para cambiar un número de signo bastará hallar su complementario.

Por ejemplo +3 es 00000011 y −3 será su complementario: 11111100 que como tiene el bit 7 a uno vemos que es negativo.

Pero este sistema tampoco funciona. Si hacemos:

(+3) 00000011
(−3) 11111100
11111111 da −127 en lugar de cero.

Este tipo de complementario se llama complemento a uno. El sistema que funciona realmente es el que emplea el **complemento a dos**. El complemento a dos de un número binario se obtiene hallando primero el complemento a uno y luego sumándole uno, sea cual sea el signo del número. Veamos un ejemplo:

(+7) 00000111
11111000 es su complementario a uno
00000001 le sumamos uno
11111001 será (−7) con este sistema.

Si ahora sumamos +7 y −7 el resultado deberá ser cero:

(+7) 00000111
(−7) 11111001
00000000 da realmenter cero.

Del mismo modo:

(+5) 00000101
(−7) 11111001
11111110 es (−2)

Veamos si es cierto:

11111110 es (−2)?
00000001 hallamos su complemento a uno
00000001 y le sumamos uno para hallar el complemento a 2
00000010 el resultado es +2, luego el número de partida era −2.

Esta es una parte un tanto árida y si alguien no tiene mucho interés en esto o no termina de entenderlo que no se preocupe, porque no es estrictamente necesario para programar en CM, aunque siempre es mejor saber cómo y por qué se hacen las cosas en lugar de emplear «recetas mágicas». Al final del libro hay una tabla y su representación en hexadecimal en complemento a dos, para evitar estos cálculos.

CAPÍTULO 2

BASIC Y CODIGO MAQUINA

2.1. Que son el ZX81 y el Z80

Es de suponer que la mayoría de lectores estará familiarizada con el ZX81. Lo que quizá ya no esté tan claro para algunos es en qué consiste y qué hace el microprocesador Z80 —o su versión más rápida Z80A—, al que nos referiremos ampliamente en lo sucesivo.

Z80A es el nombre del **microprocesador** que contiene el ZX81, y cuyo trabajo es realizar la mayor parte de las funciones y organizar el resto de las partes que componen el ZX81.

En la página 162 del Manual de Instrucciones de SINCLAIR aparece una foto del interior del ZX81 y una sucinta explicación de sus componentes. A nosotros, para programar en CM nos interesa una visión un tanto más dinámica del asunto, para tener así una idea clara de lo que puede interferir o ayudar en nuestros propósitos en un momento dado.

Podemos dividir el ZX81 en cuatro grandes bloques:

- Microprocesador Z80A
- Memoria (RAM y ROM)
- Chip de Lógica
- Comunicaciones con el exterior (Teclado, Modulador UHF, conexiones al cassette).

De estos bloques, el microprocesador y el Chip de Lógica realizan un trabajo activo durante el funcionamiento, mientras los otros

bloques tienen una función de tipo más pasivo. Esto quiere decir que el único que puede darnos problemas en un programa es el Chip de Lógica.

Veamos cada uno de estos bloques con un poco más de detalle, a través de la figura 1.

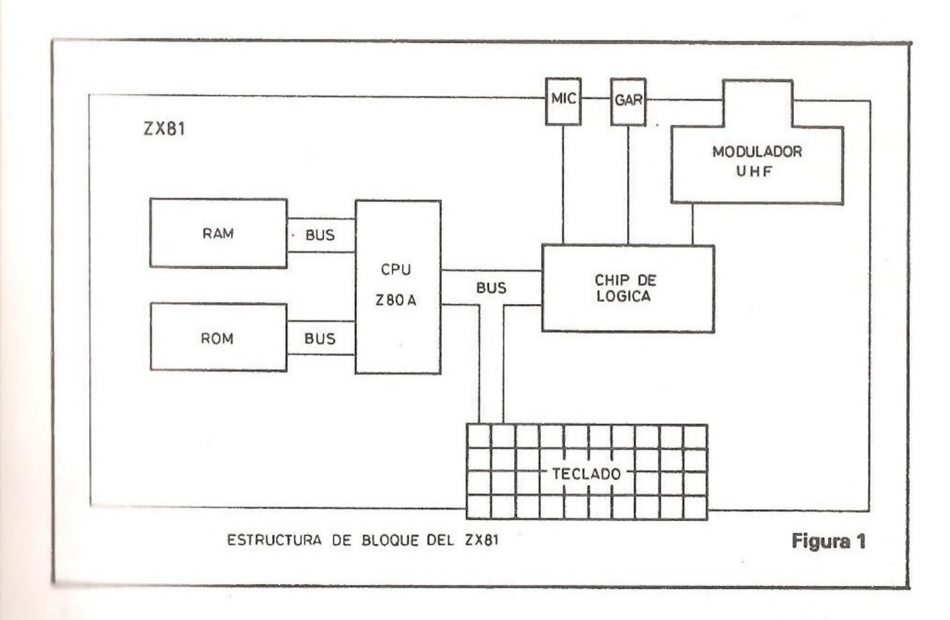


Figura 1

1. *El microprocesador Z80A*

Es el corazón de todo el sistema, del que depende de alguna forma todo el trabajo que realice el ZX81. Está conectado, directa o indirectamente, a cada uno de los restantes bloques en que hemos dividido el sistema. Las conexiones que más interesan bajo el punto de vista de este libro son las que mantiene con la memoria, ya sea RAM ó ROM.

El Z80A se comunica con la memoria mediante «buses» Un **BUS** es un conjunto de conexiones o «pistas» en paralelo —es decir, que no están interconectadas—, cada una de las cuales comunica en un momento dado un **bit** de información, esto es, un «1» ó un «0», y que actúan sincrónicamente (todas a la vez). Por ejemplo, para enviar el dato 135 dec. del bloque A al B, se conectarán ambos bloques en un momento dado, y puesto que 135 dec. en binario

es 10000111, cada una de las pistas llevará un «1» o un «0» según el siguiente esquema:

BLOQUE A	(1)	pista 7	BLOQUE B
	(0)	pista 6	
	(0)	pista 5	
	(0)	pista 4	
	(0)	pista 3	
	(1)	pista 2	
	(1)	pista 1	
	(1)	pista 0	

El Z80A se comunica con el exterior (es decir, con las otras partes del ZX81), mediante tres buses:

BUS DE DATOS: Comunica con el exterior del microprocesador la información de datos o instrucciones. Es un BUS de *8 bits,* por lo que podrá conducir números entre 0 y 255 dec. (00 y FF Hex).

BUS DE DIRECCIONES: Comunica al exterior la dirección de una posición de memoria a leer o escribir, según el caso. En el ZX81 se emplea también en parte para leer el teclado. Es un BUS de *16 bits,* por lo que puede comunicar números entre 0 y 65535 dec. (0000 y FFFF Hex). Esta es la razón por la que el ZX81 no puede manejar más de 64K de memoria de una vez (64K son 65535 bytes, o posiciones de memoria, justo el número de posiciones que puede direccionar el Z80A).

BUS DE CONTROL: Se encarga de controlar las operaciones que realiza el Z80A y de sincronizar a los diferentes elementos implicados (seleccionar RAM, autorizar el envío de datos por el BUS, etc...).

Estos tres buses comunican con otros tres buses internos del Z80A con funciones análogas, pero a nivel del propio microprocesador.

El Z80A es una versión del Z80, que sólo se diferencia de éste en que es más rápido, con 3,5 a 4 MHz frente a los 2 MHz del Z80.

El Z80A sólo entiende y trabaja en un lenguaje: el código máquina (en definitiva «1» y «0», pistas con o sin corriente), interpretando y ejecutando instrucciones contenidas en memoria, una tras otra, y archivando los resultados en lugares específicos de RAM en las ocasiones en que deba hacerlo. Para ello dispone de un «conta-

dor de programa», que contiene la dirección de memoria donde se encuentra el código de la instrucción a ejecutar.

Para realizar cada instrucción, coloca en el bus de direcciones el contenido del «contador de programa» obteniendo como resultado (por el bus de datos) el código que contiene esa dirección. Este código va a parar a un registro especial del microprocesador, donde se decodifica y se ejecuta la instrucción correspondiente al código.

En el caso de que haya que escribir un dato A en una posición de memoria B, se coloca la dirección B en el bus de direcciones, por el bus de control se envían señales que significan «escribir en RAM» y por el bus de datos se envía el dato A, que quedará copiado así en la posición B de RAM. Veremos más adelante el modo de trabajar de cada instrucción, con mucho más detalle.

2. *La memoria*

El ZX81 dispone internamente de dos tipos de memoria:

ROM: Siglas del inglés Read Only Memory (memoria de sólo lectura). Cada posición de memoria contiene un código grabado al fabricar el chip, y que por tanto es inalterable, y no puede ser modificada por el Z80A. Es una memoria del tipo llamado *no volátil,* es decir que no pierde información al desconectarse la alimentación.

Contiene una serie de rutinas en código máquina para que el ZX81 realice sus funciones (p.ej. gobernar la pantalla del TV) y para las instrucciones del BASIC, así como tablas de datos para uso del ZX81 (y nuestro, cuando las conozcamos) como los patrones de los caracteres, las instrucciones del BASIC deletreadas, etc. Parte de estas rutinas de ROM pueden ser útiles para nuestros propios fines al elaborar programas en CM.

—*RAM:* Siglas del inglés Random Access Memory (memoria de acceso aleatorio). Se trata de un tipo de memoria en la que se puede leer o escribir indistintamente. Es de tipo *volátil,* es decir, su contenido se pierde irremisiblemente al desconectar la alimentación, como seguramente el lector habrá comprobado en más de una ocasión, por desgracia.

Es en RAM donde se almacenan los programas y datos creados por el usuario y el propio programa, y también algunos datos para su uso por el ZX81, llamados *variables de sistema,* que veremos más adelante.

3. *Chip de Lógica*

Está encargado de coordinar la acción del microprocesador con el exterior en algunas funciones, como la de la codificación de la señal de pantalla que se envía al TV. Desde el punto de vista de la programación no nos interesa su funcionamiento más que en algunos puntos en que puede interferir con nuestro programa, y que veremos en su momento.

4. *Comunicaciones con el exterior*

Conectadas al Z80A mediante los buses que hemos visto, a veces con la mediación del Chip de Lógica, comunican al ZX81 con el mundo exterior. Son el teclado, salidas de cassette, salida de TV, comunicación con impresora, etc. Veremos los aspectos más importantes en capítulos específicos.

2.2. BASIC es código máquina

El microprocesador Z80, el corazón del ZX81, sólo entiende un lenguaje de programación: el binario o código máquina, para el que ha sido construido. Cualquier otro lenguaje (BASIC, Forth, Pascal...) no es en definitiva más que una traducción a CM de las instrucciones de que conste. Esta «traducción» es en realidad una serie de rutinas CM que tienen como resultado el efecto que indica la instrucción del lenguaje superior. El conjunto de estas rutinas en el caso del ZX81 reside en la ROM, de la que ocupa la mayor parte.

Por ejemplo, la sentencia BASIC PRINT "A" en el momento de ser ejecutada el intérprete transfiere el control del proceso a una rutina de ROM que empieza en la posición de memoria $0808 que se encarga de comprobar que en la posición de pantalla hay espacio para el caracter, y colocar los códigos en los lugares adecuados para que aparezca una «A» en la posición correcta de la pantalla, devolviendo cuando ha terminado el control al intérprete, que se encarga de buscar la siguiente sentencia o de emitir el mensaje correspondiente si el programa ha terminado.

Esto representa una comodidad para el programador pero tiene algunos inconvenientes:

—Se pierde una cantidad apreciable de tiempo buscando las

sentencias BASIC, decodificándolas y buscando las rutinas adecuadas en la ROM.

—Sólo se puede hacer lo que permiten las rutinas standard que corresponden al juego de instrucciones BASIC. Por ejemplo, en BASIC no se puede hacer un SCROLL hacia la derecha porque no hay ninguna instrucción BASIC para ello.

Programando en código máquina el trabajo es un poco más laborioso pero las únicas limitaciones son las que impone la estructura del hardware, con lo que las posibilidades, la rapidez y la eficacia aumentan notablemente. Además, siempre podemos aprovechar las rutinas de la ROM cuando convengan a nuestros propósitos.

2.3. Mapa de memoria del ZX81

Hemos visto que la memoria del ZX81 está formada por una serie de posiciones de memoria o bytes, cada uno de ellos con 8 bits, y que a cada una de estas posiciones se la identifica con un número de orden, desde 0 hasta 65535.

En principio, cada una de estas posiciones de memoria puede contener un código que corresponda a una variable, al programa, a información de pantalla, etc., pero por razones prácticas la memoria se divide en bloques lógicos (no físicos). Cada bloque contiene toda la información con unas características comunes. Así habrá un bloque donde estarán archivados todos los códigos del programa BASIC, otro que contendrá todas las variables BASIC, otro con la información a presentar en pantalla, etc.

Algunos de estos bloques tienen una longitud fija, pero en el ZX81 la mayor parte son de longitud variable, en función de la cantidad de información que soporten. De este modo, cuando hacemos "LET A = 1", el ZX81 aumenta la longitud del bloque que contiene las variables BASIC y coloca allí el nombre de la variable y su valor.

Cada uno de estos bloques empezará en una dirección de memoria determinada, y terminará en otra igualmente específica. Un diagrama o una lista que contenga todos los bloques en que se divide la memoria y sus respectivas direcciones de principio y final, se llama un *MAPA DE MEMORIA.*

En el ZX81 los diferentes bloques de memoria, por orden de dirección de menor a mayor, son:

— ROM
— COPIA FANTASMA DE ROM
— VARIABLES DEL SISTEMA
— PROGRAMA BASIC
— ARCHIVO DE PANTALLA
— VARIABLES BASIC
— LINEA DE ENTRADA
— PILA («STACK») DEL CALCULADOR
— ZONA LIBRE
— PILA («STACK») DEL Z80
— PILA («STACK») DE «GOSUB»
— ZONA DE USUARIO

El diagrama del mapa de memoria del ZX81 se encuentra en la página 171 del Manual de Instrucciones de SINCLAIR. En la versión española del Manual hay un *error importante* en el diagrama: las zonas VARIABLES y MEMORIA DE PANTALLA están intercambiadas, dando una imagen falsa de la distribución de memoria del ZX81. El orden correcto es el que se da en este libro; es decir, primero ARCHIVO DE PANTALLA y a continuación VARIABLES BASIC.

Veamos qué contienen exactamente cada uno de estos bloques:

ROM: Contiene las rutinas (en CM), datos e información para que el ZX81 pueda trabajar en BASIC. Su longitud y posición son fijas, entre 0 y 8192 dec.

COPIA FANTASMA DE ROM: Ocupa desde 8193 hasta 16384 dec. En realidad es un espacio vacío, sin ninguna memoria asignada, pero por la forma en que está construido el ZX81 se comporta como si hubiera una copia de la ROM a continuación de la verdadera.

Para poder disponer de este espacio libre para proyectos y ampliaciones (por ejemplo un módulo de memoria RAM de 64K), hay que realizar circuitos de decodificación y direccionamiento especiales. Esta zona no es accesible por software. Por cierto que las memorias de 64K para el ZX81, en realidad no proporcionan más de 56K, porque se necesitan direcciones para los 8K de ROM y hemos visto que el Z80 sólo puede direccionar 64K en total y como máximo.

VARIABLES DEL SISTEMA: Es otra zona de dirección y longitud fijas que ocupa desde 16384 a 16508. Contiene información para uso

interno del ZX81. Parte de esta información corresponde a direcciones de memoria donde empiezan o terminan otros bloques del mapa de memoria. En el apartado 2.4 de este libro y en las páginas 177 a 180 del Manual de Instrucciones de SINCLAIR se da información exhaustiva sobre estas variables de sistema.

PROGRAMA BASIC: Esta zona empieza en la posición 16509 y su longitud depende de la extensión del programa BASIC que exista en el ZX81 en un momento dado, pues es aquí donde se almacena.

ARCHIVO DE PANTALLA: Empieza a continuación del bloque PROGRAMA BASIC y por tanto la dirección dependerá de la longitud de éste. Esta dirección la coloca el ZX81 automáticamente en la variable de sistema D-FILE (ver apartado 2.4). Sobre la longitud de este bloque, véase el Capítulo 15.

Contiene toda la información que se presenta en la pantalla del TV en un momento dado, en forma de los códigos de los caracteres a representar, más unos códigos de fin de línea.

VARIABLES BASIC: Empieza a continuación del bloque ARCHIVO DE PANTALLA, y la dirección de inicio está en la variable de sistema VARS (ver apartado 2.4). Contiene las variables BASIC, una detrás de otra, y por orden de asignación. Como consecuencia, su longitud dependerá de las variables que tenga almacenadas el ZX81 en un momento dado.

LINEA DE ENTRADA: Empieza a continuación de VARIABLES BASIC. Contiene la información que se está entrando por el teclado antes de pulsar NEWLINE, es decir, lo que aparece en las dos líneas inferiores de la pantalla, aunque esta zona no tenga nada que ver con la presentación en pantalla.

Por ejemplo, al ir escribiendo una nueva línea de programa, ésta se va almacenando en esta zona, que crece cada vez que se pulsa una tecla. Al pulsar NEWLINE, si no hay errores de sintaxis, la zona se copia en el lugar adecuado del bloque PROGRAMA BASIC, que previamente se habrá expandido en la longitud necesaria. La dirección donde empieza esta zona está indicada en la variable de sistema E—LINE.

PILA («STACK») DEL CALCULADOR: Empieza a continuación de LINEA DE ENTRADA. Una pila («stack») es una estructura especial para almacenar datos temporalmente (ver Capítulo 10), y cuya

longitud depende de lo que se tenga almacenado. La pila («stack») del calculador se emplea para almacenar números y resultados parciales cuando se realizan cálculos en BASIC. Por ejemplo, para calcular $(3+2)\times(5+7)$ se calcula $5+7$ y se almacena 12 en la pila («stack») del calculador; se calcula $3+2$ y el resultado, 5, se multiplica con el resultado parcial almacenado en la pila («stack»), dando 60.

La dirección donde empieza la PILA (el «STACK») DEL CALCULADOR se almacena en la variable de sistema STKBOT y la del final en STKEND.

ZONA LIBRE: Es la zona extra de RAM que queda libre, sin ningún uso. Su longitud y posición dependen de las de las otras zonas, es decir, de que el ordenador esté más o menos lleno.

Hasta PILA («STACK») DEL CALCULADOR inclusive los diferentes bloques de RAM están uno a continuación de otro, y si se reduce uno de ellos, todos los que están por encima de él, se desplazan hacia direcciones de memoria menores. Podríamos decir que están unidos y fijados por el principio del «superbloque» que forman.

Los bloques que se describen a continuación también son contiguos, pero fijados por el final, de forma que si reducimos uno de ellos, todos los que están por debajo (hasta la zona libre) se desplazarán hacia direcciones de memoria mayores. Estos bloques son:

PILA («STACK») DEL Z80: Es otra estructura en pila («stack»), para uso del Z80, muy empleada en programas en CM. Esta zona, de longitud variable, está situada inmediatamente antes de la PILA («STACK») DE «GOSUB». Su dirección se almacena en un registro especial del Z80. Para más detalles sobre su funcionamiento y su uso, ver los Capítulos 9 y 18.

PILA («STACK») DE «GOSUB»: Está situada entre la PILA («STACK») DEL Z80 y la ZONA DE USUARIO, delimitada por las direcciones que se almacenan en las variables de sistema ERR—SP y RAMTOP respectivamente. Es otra estructura en pila («stack»), que almacena los números de línea a los que deberá volver el programa BASIC cuando encuentra la instrucción RETURN en una subrutina, para volver a la secuencia principal.

ZONA DE USUARIO: Zona comprendida entre la variable de sistema RAMTOP y la dirección de memoria mayor disponible, es decir, el final de la memoria.

Cuando se conecta el ZX81, la variable RAMTOP se coloca automáticamente indicando la primera dirección de RAM que ya no existe (el final de RAM), por lo que la zona tiene longitud cero.

Para poder disponer de esta zona, hay que cambiar el valor de RAMTOP de forma que deje una zona de memoria entre ésta y el final de RAM. (Ver en el capítulo 26 del Manual SINCLAIR la forma de hacer esto). Esta zona no se borra cuando se emplea la instrucción NEW, ni puede ser alcanzada por el BASIC.

Resumiendo, estas zonas no son bloques separados físicamente unos de otros, sino la estructura lógica en que se organiza la memoria del ZX81. Los bloques van creciendo de forma que cuando el ordenador está lleno, la PILA («STACK») DEL CALCULADOR y la PILA («STACK») DEL Z80 se encuentran en una zona más o menos central de RAM.

2.4. Variables del sistema

Hay una serie de bytes, desde las posiciones 16384 a 16508 inclusive, que son reservados por el ZX81 para soportar información de uso interno y específico, relacionada con su funcionamiento. No son variables BASIC ni nada parecido, y su longitud oscila de uno a 33 bytes, según la variable de que se trate.

En el capítulo 28 del Manual de SINCLAIR se da una lista detallada de estas variables de sistema, con una pequeña explicación de la función de cada una.

Por el tipo de función que realizan, las podemos clasificar en las siguientes categorías:

- Punteros separadores de bloques lógicos de RAM, todas de 2 bytes: D—FILE, VARS, E—LINE, STKBOT, STKEND, ERR—SP, RAMTOP.

- Punteros dentro de una zona de RAM que proporcionan información al intérprete BASIC. Todas de 2 bytes: DF—CC, DEST, CH—ADD, X—PTR, NXTLIN, T—ADDR.

- Indicadores en «notación BASIC» que proporcionan información al intérprete, de 1 ó 2 bytes: PPC, E—PPC, S—TOP, OLDPPC, COORDS, S—POSN.

- Indicadores para uso interno, sobre características del propio sistema en un momento dado (1 ó 2 bytes): ERR—NR, MODE,

VERSN, BERG, MEM, DF—SZ, LAST—K, MARGIN, STRLEN, SEED, FRAMES, PR—CC.

—Flags de control de estados internos, 1 byte: FLAGS, FLAGX, CDFLAG.

—Zonas acotadas para soportar información: PRBUFF (33 bytes), MEMBOT (30 bytes).

La función de los punteros separadores de bloques lógicos de RAM se describe en el apartado 2.3. Si se cambian las direcciones que contienen, el Z80, y por tanto el ZX81, creerá que la zona determinada empieza en esa nueva dirección, lo que puede ser útil en algunos casos. Por ejemplo, si construimos una copia de la estructura del archivo de pantalla (ver Capítulo 14) en otra zona de RAM, y en un momento dado colocamos la dirección donde empieza esta estructura en la variable de sistema D—FILE, el ZX81 creerá que el ARCHIVO DE PANTALLA empieza ahí, y lo que aparecerá en la pantalla del TV será esta nueva estructura en lugar del auténtico ARCHIVO DE PANTALLA. Esto nos permitirá disponer de dos (o más) «pantallas» intercambiables de forma prácticamente instantánea.

Las variables del sistema cuyo objeto es proporcionar información al intérprete BASIC, en general tienen muy poco interés cuando se trabaja en CM, porque en este caso el intérprete no se utiliza.

Las zonas acotadas para soportar información tienen interés porque son bytes de los que disponemos en determinadas condiciones, para situar las «variables internas» de nuestro programa en CM.

PRBUFF tiene 33 bytes (32 libres y el 33 con código 118 dec.). Es un tampón donde se copia la línea que se esta imprimiendo por la impresora cuando esta está activa. Si no vamos a usar impresora en nuestro programa en CM, disponemos de 32 bytes libres para lo que queramos. Hay que tener en cuenta que al pulsar NEWLINE estos 32 bytes son puestos automáticamente a cero por el ZX81, con lo que se perderá la información que pudieran contener.

MEMBOT tiene 30 bytes, y es un área empleada en cálculos en BASIC. Si no se realizan cálculos con variables BASIC, estos 30 bytes están libres para nuestro uso.

Cuando el problema de falta de memoria se hace crítico, como en el caso de disponer sólo de 1K de RAM, estos pequeños espacios libres pueden ser vitales.

Conociendo y empleando adecuadamente las variables de sistema, se pueden conseguir incrementos notables en la flexibilidad y potencia del ZX81, pues se puede adaptar mejor el ordenador a cada problema específico.

2.5. Localización de los programas

Un programa no es más que una sucesión de códigos de instrucciones, almacenados en posiciones contiguas de memoria, y en principio dispuestos en el mismo orden en que serán ejecutados. En CM no existen ni las sentencias ni los números de línea: el equivalente a número de línea viene a ser en CM la posición de memoria.

En BASIC hemos visto que hay una zona extensible reservada en RAM donde se almacena el programa, que va desde 16509 hasta la posición anterior a la indicada por la variable de sistema D—FILE. En CM no existe nada parecido a esta estructura y por tanto un programa podrá residir en cualquier zona de RAM que no vaya a ser borrada o reescrita durante el funcionamiento.

Según la estructura del programa, su función, su tamaño, serán mejores unos u otros lugares de entre todos los posibles. Un programa estará tanto mejor localizado cuanto mejor cumpla algunas de las siguientes características:

—Estar fijo en unas posiciones de memoria determinadas, sin ser desplazado arriba y abajo de RAM, puesto que si se desplaza cambian las direcciones de memoria («los números de línea»), haciendo muy difícil la operatividad de algunas instrucciones (de saltos, por ejemplo).

—Estar a salvo de borrados durante el funcionamiento. Hay que tener en cuenta que la zona de memoria se expande y contrae según las necesidades, por otra parte previsibles, del ZX81.

—Que sea fácil de cargar hacia y desde el cassette. Una zona libre intermedia de RAM puede ser muy estable y estar a salvo de borrados, pero cada vez que se necesite el programa habrá que entrarlo desde el teclado (lo que podría ser muy engorroso), porque esta zona no se puede cargar desde el cassette.

Analicemos ahora algunas posibles zonas donde puede residir un programa en CM, que cumplan estas condiciones:

1. *En líneas REM:* Las líneas REM son ignoradas por el intérprete BASIC, por lo que no interferirán si usamos rutinas en CM

alojadas en líneas REM como parte de programas mayores en BASIC. Además no cambian de localización en RAM a menos que modifiquemos líneas anteriores del listado. Por otra parte al formar parte del «programa en BASIC» (aunque éste no conste más que de líneas REM con CM), se grabarán y cargarán del cassette como si fueran un programa normal.

Para colocar un programa en CM en una línea REM hay que construir primero la línea con tantos caracteres como bytes tenga el programa, por lo menos, y luego ir sustituyendo los caracteres por los códigos CM de alguna forma. Si hacemos que la línea REM sea la primera del programa (por ej. 1 REM) tenemos además la ventaja de que siempre empieza en la misma dirección de memoria: 16514 es la dirección del primer byte del texto de la línea.

En el caso de que una vez introducido el programa en CM se quiera modificar la longitud de la línea REM, hay un factor muy importante a tener en cuenta: si uno de los códigos introducidos es $3E, no se puede modificar la longitud de la línea editándola. Esto es así porque este código lo emplea el ZX81 para indicar que los 5 bytes que siguen son un número en «notación SINCLAIR», y cuando lista la línea en pantalla o la edita los omite, por lo que si editamos una línea que contenga en alguna parte códigos $3E, perderemos bytes irremisiblemente, y el ZX81 se confundirá.

Por desgracia, el código $3E corresponde a la instrucción "LD A, (HL)", una de las más usadas en CM. Por tanto es mejor no tocar las líneas REM con CM si no se está muy seguro.

Una forma de añadir caracteres a una línea REM que sea la primera del programa es la siguiente:

—Crear una línea "2 REM" con tantos caracteres como bytes se necesiten menos 6.

—Añadir las siguientes líneas en cualquier parte del programa (al final, por ej.) y hacer un GOTO a la primera de estas líneas:

```
LET L = 16515 + PEEK 16511 + 256 * PEEK 16512
LET L = L + PEEK L + 256 * PEEK(L + 1) – 16511
POKE 16511,L – 256 * INT(L/256)
POKE 16512,INT(L/256)
```

Se pueden entrar también como comandos, en cuyo caso no se puede hacer LIST hasta después de entrar la última.

Entrar LIST: Aunque aparentemente no ha ocurrido nada, la segunda línea se ha añadido como continuación de la primera, lo que

puede comprobarse con "LIST 2", que no listará la línea 2, sino a partir de la siguiente, si existe.

Del mismo modo, para acortar una línea REM que sea la primera del programa, hay que entrar las siguientes líneas:

```
LET A= a la dirección del último BYTE que queramos con-
  servar en la línea "1 REM"
LET B=A–16511
LET C = PEEK 16511 + 256 * PEEK 16512–B–4
POKE 16511, B–256 * INT (B/256)
POKE 16512,INT(B/256)
POKE A+1,118
POKE A+2,0
POKE A+3,2
POKE A + 4,C–256 * INT (C/256)
POKE A+5,INT(C/256)
POKE A+6,234
```

Con esta rutina se puede acortar una línea en un mínimo de 6 bytes. Hacer GOTO a la primera de estas líneas, y luego LIST. Aparecerá una nueva línea "2 REM" con los caracteres eliminados de la primera; puede eliminarse entrando el número de línea «2».

Con estas rutinas se pueden construir líneas REM tan largas como se quiera (de 4 ó 5K, por ejemplo), que ocuparán más de una pantalla si se intenta listarlas. En este caso hay que ir con cuidado, porque el ZX81 puede entrar en un ciclo sin fin en el que intenta listar la línea, y al no conseguirlo borra pantalla y empieza otra vez. Así hasta que se corte la alimentación, con lo que se perderá todo el trabajo realizado. Para evitar esto, no emplear LIST a secas, sino «LIST 1», con lo que al llenar la pantalla, el ZX81 se detiene con el error «5/0». Tampoco hay que eliminar la línea siguiente a la REM, porque producirá los mismos efectos. En su lugar lo mejor es sustituirla por una línea REM vacía (sin caracteres).

2. *En una variable:* Otro lugar donde se puede colocar un programa en CM es en la zona de variables, formando parte de una variable de cadena, del tipo A$. Tiene la ventaja de que se pueden almacenar en cinta, y además no interfiere en las operaciones del listado, pero por otra parte, la zona de variables cambia de posición en función de la longitud del programa BASIC, lo que en ocasiones puede representar un serio inconveniente.

Para reservar espacio en esta zona, primero hay que dimensionar

una variable de cadena con tantos caracteres como bytes vaya a ocupar el programa en CM, con el comando DIM A$, por ejemplo. Esta variable que contendrá el CM debe ser la primera definida después de CLEAR, o RUN, para que esté en primer lugar en el archivo y sea fácilmente localizable. La posición del primer byte de la variable se calcula mediante PEEK 16400 + 256 * PEEK 16401 + 6.

Otro inconveniente de utilizar una variable como soporte de CM es que si se emplea RUN o CLEAR, o se redimensiona la variable, se pierde su contenido de forma definitiva.

3. *Tras RAMTOP:* Este es otro sitio donde almacenar CM, que mantiene su posición en RAM fija, y está a salvo de borrados accidentales (incluso NEW), pero tiene la desventaja de que no puede comunicar directamente con el cassette.

Otra ventaja importante de los programas situados tras la variable de sistema RAMTOP es que pueden trabajar con cualquier programa BASIC, o CM, situado en el lugar habitual (zona PROGRAMA de RAM), manteniéndose totalmente independiente de él. Para ello hay que cargar primero la rutina en CM, instalada por ej. en «1 REM», luego trasladarla tras RAMTOP, y luego cargar el segundo programa normalmente, teniendo en cuenta que el programa en CM debe estar diseñado para funcionar en las posiciones de memoria tras RAMTOP, y no en las que se almacena (líneas REM, por ej.).

4. *En zonas libres de RAM:* En principio se puede almacenar CM también en posiciones de memoria RAM que no están controladas por ninguna variable de sistema, y que por tanto estarán en la zona libre intermedia de RAM. Sin embargo, esta zona no cumple ninguna de las condiciones expuestas anteriormente, por lo que su uso sólo es aconsejable en casos muy particulares, empleando para cargar el CM un sistema análogo al del apartado anterior.

2.6. Mnemónicos y código objeto

El ZX81, y por extensión cualquier ordenador, trabaja realmente en binario, interpretando y efectuando códigos de «1» y «0», que corresponden a las distintas instrucciones que el constructor del microprocesador ha dispuesto al fabricarlo. Por ejemplo, el código 000000 bin. corresponde a una instrucción que no tiene ningún efecto, y que se emplea únicamente para perder tiempo (algo parecido al «PAUSE» del BASIC).

Es importante recordar que este es el único tipo de información que el Z80 entiende y es capaz de procesar. Cualquier otro sistema (mnemónicos, lenguajes de alto nivel, etc.) no son sino «traducciones» de estos códigos o rutinas formadas con ellos para que el funcionamiento del microprocesador, y en definitiva del ordenador, se parezca un poco más al modo de pensar de los humanos.

Es evidente que sería muy engorroso tener que programar directamente en binario recordando el conjunto de unos y ceros que tiene asignado cada instrucción (aunque los primeros ordenadores funcionaban únicamente con este sistema).

Un primer paso consiste en emplear el sistema hexadecimal en lugar del sistema binario. De esta forma es más fácil recordar un código C9 que su equivalente 11001001 binario. De todas formas, el Z80 sigue trabajando en binario y la traducción es sólo para nosotros.

Con la notación hexadecimal sigue siendo una tarea ardua programar y descifrar un listado en CM. Por ello se recurre al empleo de *«mnemónicos»,* que son agrupaciones de letras que recuerdan la función que realiza la instrucción a la que se asignan. A cada código del juego de instrucciones le corresponde un mnemónico y sólo uno. Por ejemplo a la instrucción que hemos visto antes, con código 00 hex. le corresponde el mnemónico NOP (no operation). Se puede ver una lista completa de los códigos y sus mnemónicos en el Apéndice I.

Para que podamos trabajar directamente con mnemónicos se necesita un programa que los traduzca a sus códigos respectivos, y que recibe el nombre de PROGRAMA ENSAMBLADOR. Si no se dispone de este programa, hay que buscar los códigos en una tabla y entrar estos en el ZX81, no los mnemónicos.

2.7. Rutina sencilla para entrar CM

La rutina que se describe a continuación sirve para entrar CM en forma de códigos hexadecimales, en una línea 1 REM preparada como se indica en el apartado 2.5. Se puede entrar varios códigos a la vez escribiéndolos uno a continuación de otro, sin espacios, antes de pulsar NEWLINE. Una vez entrados todos los códigos hay que pulsar EDIT y entrar STOP, para detener la rutina de carga.

La variable X de la línea 10 contiene la dirección de la primera

posición de memoria en la que se efectuará la carga. Una vez entrado el CM, la rutina de carga se puede eliminar del ZX81 entrando uno a uno los números de línea 10 al 60. No se debe emplear NEW porque se perderá también el CM.

```
 1 REM... caracteres cualesquiera, soporte del CM...
 5 LET A$ " "
10 LET X=VAL "16514"
20 IF A$="" THEN INPUT A$
30 POKE X, CODE A$ * CODE "(" + CODE A$(2)—VAL "476"
40 LET A$= A$(INT PI TO)
50 LET X=X + SGN PI
60 GOTO CODE "="
```

Este listado puede parecer un tanto raro a primera vista. Se debe a que está compactado al máximo para dejar más espacio al CM. Su funcionamiento se verá más claro si se tiene en cuenta que:

```
VAL "16514"  = 16514
CODE "("     = 16
VAL "476"    = 476
INT PI       = 3
CODE "="     = 20
SGN PI       = 1
```


CAPÍTULO 3

LOS REGISTROS

3.1. Qué es un registro: características

Hemos visto que el microprocesador Z80 comunica con el exterior mediante BUSES (ver apartado 2.1). Una vez la información llega al microprocesador deberá situarse en algún sitio donde pueda almacenarse temporalmente y ser tratada. Estos lugares del Z80 que contienen información son los *REGISTROS*. En un sentido muy amplio, un registro se comporta como una variable numérica del BASIC; es decir, podemos asignar un número a un registro, operar con él, y copiarlo en otro registro, de un modo análogo a como se hace en BASIC con una variable. Por otra parte hay bastantes diferencias entre una variable BASIC y un registro.

1. Mientras en BASIC podemos definir tantas variables como queramos y nos permita la memoria disponible, en CM estamos limitados a unos pocos registros definidos por el constructor del Z80, que se designan por las letras A,B,C,D,E,H y L. El Z80 tiene algunos otros registros más particulares, que veremos más adelante. Por tanto estamos limitados en principio a 7 «variables».

2. En BASIC una variable suele almacenar un dato durante todo o gran parte del proceso. En CM, como disponemos de pocos registros, sólo se traen los datos a los registros cuando deben sufrir algún cambio, y una vez efectuado éste se devuelven a posiciones de memoria RAM que habremos determinado, que se encar-

gan de almacenar el dato hasta que se requiera una nueva modificación o consulta.

3. Los registros de los que estamos hablando tienen *8 bits* (1 byte), por lo que *sólo pueden contener un número entero positivo entre 0 y 255.* En el Capítulo 2 hemos visto cómo tratar números con signo.

4. Cuando a una variable BASIC se le asigna un número mayor que el que puede contener (10^{+39}), el programa se detiene con error 6/número de línea. Cuando a un registro se le intenta asignar un valor mayor que 255, sencillamente se vuelve a poner a cero, y sigue desde ahí. Así si intentamos hacer 255 + 1 da 0 en lugar de 256, 255 + 4 da 3, 250 + 62 da 56, etc.

Una imagen intuitiva de esto podría ser una rueda con marcas numeradas desde 0 a 255, ambas inclusive (como los botones de combinación de una caja fuerte), de modo que los números 255 y 0 estén contiguos.

En resumen, un REGISTRO no es más que una posición de memoria RAM construida en el interior del microprocesador, que se designa con una letra en lugar de un número, sobre la que trabaja la mayor parte de las instrucciones del microprocesador y que puede intercambiar información con otros bloques del ordenador.

3.2. Registros internos

Además de los siete registros A,B,C,D,E,H y L que se utilizan para tratar datos, el Z80 dispone de otros registros para uso interno, algunos accesibles al programador y otros no, que son una especie de variables de sistema del microprocesador. Estos registros son:

REGISTRO INS: Es de 8 bits, y almacena el código de la instrucción que se va a ejecutar para decodificarla. No es accesible al usuario, y trabaja automáticamente.

REGISTRO R: Es de 8 bits. Es un contador de tiempo que va decrementando automáticamente. Se emplea para la operación de refresco de las memorias RAM de tipo dinámico. En este tipo de memoria la información se pierde al cabo de unos milisegundos si no es reescrita de nuevo. Esta operación la realiza el Z80 de forma automática, y para el control de tiempo emplea el registro R.

El programador podría usarlo como «reloj», si no fuera un «registro prohibido» en el ZX81, como veremos más adelante.

REGISTRO I: De 8 bits, se emplea en interrupciones, llamadas de tipo 2, donde este registro proporciona el byte más significativo de una dirección que almacena la verdadera dirección a la que salta el programa cuando sufre una interrupción de este tipo. Que nadie se preocupe si no entiende nada de este párrafo, porque el registro I es uno de los «prohibidos» en el ZX81, y podemos olvidarnos de él.

REGISTRO SP: Tiene 16 bits. Contiene la dirección de memoria RAM donde está la cima de la pila («stack») del Z80, estructura que se describe en el Capítulo 9. Es accesible por programa.

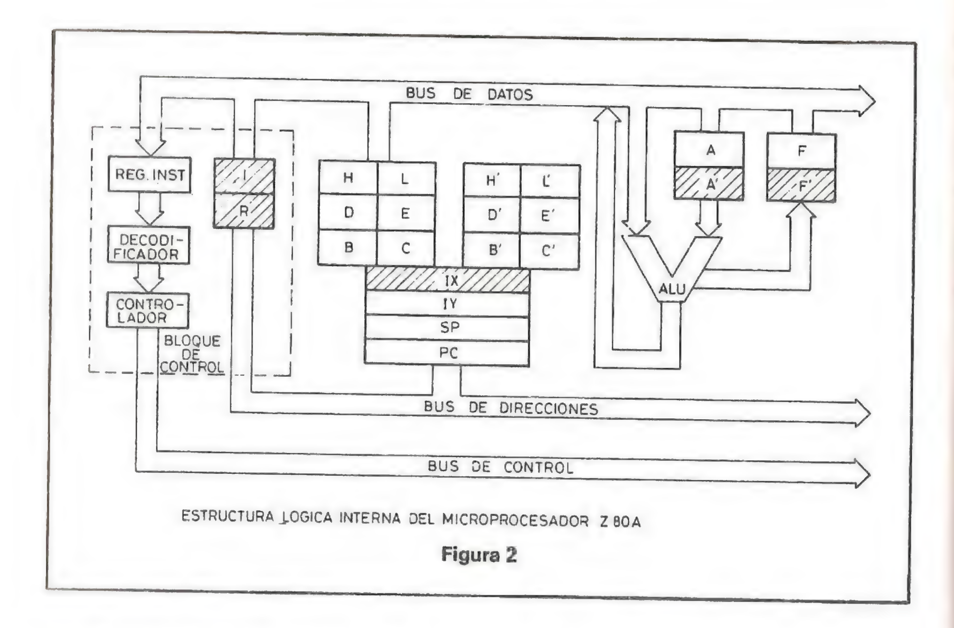
REGISTRO PC: Tiene 16 bits. Contiene la dirección de memoria donde está la siguiente instrucción a ejecutar por el programa. Su contenido se incrementa en 1 automáticamente después de ejecutar cada instrucción. Se puede modificar por programa, aunque no es frecuente hacerlo.

REGISTRO F: Aunque es accesible por programa, y además se usa con mucha frecuencia, se puede considerar un registro interno. Tiene 8 bits, que se interpretan independientemente unos de otros, llamados FLAGS. No son más que indicadores de si se cumplen determinadas condiciones o situaciones, que se describirán con detalle en el Capítulo 7.

Para facilitar una comprensión más clara de cómo funcionan estos registros, se incluye la figura 2, que corresponde a la disposición *lógica* (físicamente están distribuidos de otra forma dentro del chip) de los registros del Z80, y de cómo están unidos por los BUSES internos.

3.3. Parejas de registros

El Z80 dispone de la facilidad de manejar registros de 8 bits en parejas determinadas, como si fueran registros mayores, de 16 bits, lo que permite manejar números entre 0 y 65535 dec. en un solo bloque. Las parejas formadas se designan por las iniciales de los dos registros que las forman. Son las siguientes: BC, DE, HL y AF. Puesto que el registro F es un tanto especial, el par AF sólo se usa como tal en casos muy específicos. De los



ESTRUCTURA LOGICA INTERNA DEL MICROPROCESADOR Z 80A

Figura 2

otros tres, el más versátil es el par HL. Los registros son considerados por el Z80 aisladamente o en parejas según la instrucción que se emplee.

3.4. Banco alternativo de registros

Además del juego de registros A, F, B, C, D, E, H y L, el Z80 dispone de otro banco de registros que se designan con A', F', B', C', D', E', H', L'. Sólo se puede trabajar con uno de los bancos a un tiempo (en realidad sólo con A, F, B, C, D, E, H, L) pero, en un momento dado, pueden intercambiarse los contenidos de cada registro con su registro alternativo. Esto puede ser útil en determinada situaciones, pero en otras los registros «primas» son «registros prohibidos» en el ZX81, como se describe en el siguiente apartado.

3.5. Registros prohibidos en el ZX81

Hablando en términos estrictos no hay ningún registro absolutamente prohibido al programar el ZX81 en CM. Lo que ocurre es que algunos registros, bien por la estructura del hardware, bien porque los utiliza la rutina de DISPLAY que interrumpe nuestro pro-

grama 50 veces por segundo, o no es posible utilizarlos por nosotros en la forma en que se pueden usar en otros ordenadores, o sólo se pueden emplear en los momentos y situaciones en que no vayan a ser arrebatados por la rutina de DISPLAY, dejándolos en todo caso otra vez con sus valores originales.

Hechas estas precisiones, veamos cuáles son estos registros y en qué forma se ven afectados:

REGISTRO I: Aunque normalmente se emplea para suministrar parte de direcciones en las interrupciones, en el ZX81 contiene el byte más significativo de la dirección del inicio de la tabla que hay en ROM que contiene los dibujos de los caracteres que se pueden representar, en forma de ocho bytes por carácter, donde cada byte es una línea y cada bit dentro del byte puede ser uno si es un punto negro o cero si es un punto blanco. La dirección contenida en el registro I es $1E. En teoría parece que si cargamos I con un dato que dé una dirección en RAM, podríamos definir nuestros propios caracteres construyendo una tabla parecida a la de ROM. Desgraciadamente esto no es posible por razones técnicas y si se intenta esto, la pantalla aparece totalmente negra. La única aplicación que he encontrado para ello es que, cargando este registro con una dirección de RAM y luego con $1E varias veces, separadas por un lazo de retardo (y dejándolo al final con $1E) se produce un efecto de «explosión» bastante interesante.

PAR DE REGISTROS A'F': El par A «prima» F «prima» forma parte del banco de registros alternativos, y en SLOW se emplea para contar el número de líneas en blanco de los márgenes superior e inferior de la pantalla. Por tanto no se puede utilizar en SLOW, porque la pantalla dará un salto parecido al del final de PAUSE y perderemos su contenido, aunque no ocurre ningún desastre.

REGISTRO IX: Se utiliza en la rutina de DISPLAY para efectuar un salto a una u otra parte de la propia rutina, en función de determinadas condiciones. Por tanto, no se puede utilizar cuando se trabaja en SLOW.

REGISTRO IY: Lo utiliza la ROM como puntero en la zona de variables de sistema cuando trabaja el intérprete BASIC. Sin alterar su valor y utilizando cuando sea posible el formato (IY + d) de las instrucciones que lo tengan, tenemos acceso a las variables de sistema sin tener que ocupar ningún otro registro como puntero. Si se altera el valor de IY, antes de volver al BASIC debe restablecerse a $4000.

CAPÍTULO 4

CODIGO MAQUINA EN EL ZX81

4.1. Por qué hay particularidades en el ZX81

Si se aprende a programar en CM con libros que no son específicos para el ZX81, resulta un tanto sorprendente que haya una serie de instrucciones, e incluso algunos registros del microprocesador Z80, QUE NO PUEDEN EMPLEARSE EN EL ZX81. Esto es debido a que, para reducir el precio de venta, gran parte de las funciones que en otros ordenadores se realizan con circuitos especiales, en el ZX81 es el microprocesador, con ayuda del Chip de Lógica, quien las lleva a cabo, a través de rutinas contenidas en la ROM.

Una de estas rutinas que se realizan por software es la gestión de la pantalla del TV, en concreto mandar una serie de impulsos digitales al modulador UHF, de modo que un 0 sea un punto blanco en la pantalla y un 1 un punto negro, más los impulsos de sincronismos vertical y horizontal.

Esto debe hacerse unas 50 veces por segundo para que la pantalla dé una imagen permanente, y para ello el Chip de Lógica manda una interrupción al Z80 en el momento adecuado, que deja lo que está haciendo y se dedica a la gestión de pantalla. Cuando ha terminado, sigue con lo que estaba haciendo hasta que sufre una nueva interrupción.

Por esta causa sólo se describirán someramente las instrucciones relacionadas con interrupciones, puesto que su uso y aplicaciones en el ZX81 requieren conocimientos específicos que exceden los propósitos de este libro. De hecho esto no supone ningún inconveniente cuando se trata de programas que no utilicen periféricos especiales, que es la inmensa mayoría de los casos.

Debido a este particular sistema para la gestión de pantalla, el ZX81 es un tanto más lento que otros ordenadores más caros. Como consecuencia, un programa en CM en el ZX81 también será más lento que en otros ordenadores, aunque seguirá siendo muchísimo más rápido que en BASIC.

4.2. La instrucción «HALT»

Esta es una instrucción que *NO DEBE USARSE NUNCA.* Tiene el efecto de detener el proceso que esté realizando el Z80 y efectuar NOP's indefinidamente, hasta que reciba una interrupción o se desconecte la alimentación.

Esta instrucción la emplea la rutina de ROM encargada de la gestión de pantalla, para sincronizarse con el Chip de Lógica, por lo que si se emplea en otro momento, el ZX81 se confunde y se «encasquilla». Su código es $76 (118 dec.), el mismo que las marcas de fin de línea del archivo de pantalla (ver Capítulo 14), y funciona deteniendo al Z80 hasta que llega el momento de empezar a modular la siguiente línea de pantalla. Este momento será indicado por una señal de interrupción mandada por el Chip de Lógica.

4.3. La instrucción «RET»

Es la instrucción análoga a «RETURN» del BASIC pero en CM. Su misión es devolver el control del programa a la secuencia principal cuando éste encuentra «RET» en una subrutina. En el ZX81 un programa en CM se trata como si fuera una subrutina de un programa BASIC, aunque éste no exista, y por tanto emplearemos «RET» al final de los programas en CM para devolver el control al BASIC, si existe, o al ZX81 en cualquier caso.

El programa en CM más corto posible sólo tiene una instrucción: RET. Lo que hace es sencillamente volver al BASIC.

La instrucción «RET» tiene código $C9. Este código, cuando es interpretado por el monitor de la ROM del ZX81, corresponde a «TAN», de modo que en los listados que da el ZX81 del CM (esas líneas llenas de caracteres aparentemente sin sentido), podemos localizar a simple vista los finales de las rutinas buscando «TAN». Esto se cumple suponiendo que alguno de los 5 bytes que precedan a «TAN» no sea $3E, puesto que si es así el código $C9 (o sea, la instrucción RET en CM o TAN en los listados) estará en bytes ocultos (ver apartado 2.5).

4.4. «USR»: cómo usarlo

Antes de entrar de lleno a describir con detalle todas y cada una de las instrucciones de que disponemos para programar el Z80, vamos a ver cómo ejecutar programas en CM en el ZX81.

Para ejecutar un programa BASIC disponemos del comando «RUN», que hace que el ordenador empiece por el principio, o de «RUN n», en cuyo caso el ZX81 empieza a correr el programa a partir de la línea de programa «n». En CM no disponemos de un comando para correr programas, sino de una función: «USR».

Para que se vea más claro la forma de usarla, la compararemos con la utilización de otra función que resulte más familiar, por ejemplo «SQR», que calcula la raíz cuadrada del número o expresión que le siga. En realidad esto lo hace una rutina en CM de la ROM, a la que llama el intérprete BASIC del ZX81 cuando encuentra «SQR». Podemos calcular la raíz cuadrada de 25, por ej., pero no podemos construir una línea como «10 SQR 25», porque el ZX81 no sabría que hacer con el resultado. Lo que sí podemos hacer es «PRINT SQR 25», «LET A= SQR 25», «RAND SQR 25», etc. En cada caso el resultado de la función, 5, se presenta en pantalla, se asigna a la variable A, se emplea para situar la secuencia de números aleatorios, etc.

La función «USR n» se comporta exactamente igual a «SQR», sólo que evidentemente no calcula raíces cuadradas. Lo que hace es hacer saltar el discurso del programa a la posición de memoria que indique el número o expresión «n», y seguir a partir de ahí en CM, hasta que encuentre una instrucción «RET». Entonces el resultado de la función es el contenido del par de registros BC, que será lo que se presente en pantalla, se asigne a una variable, etc., en función de cómo esté montado «USR» en la línea de programa («PRINT USR n», «LET A= USR n», etc.).

Así una rutina en CM situada en una línea «1 REM» que cargue el par de registros BC con el número 3000 dec., si se ejecuta con «PRINT USR 16514», dará como resultado la aparición en la pantalla de «3000». (Recordemos que el primer carácter de una línea «1 REM» está siempre en la dirección 16514).

Lo que ocurre es que en la mayoría de los casos el contenido de BC al final de la rutina no nos interesa para nada, y en cambio lo que sí nos interesa es el proceso del propio programa (un juego de «marcianitos», por ejemplo).

Entonces lo que hay que hacer es montar la función «USR» de llamada al CM de modo que el resultado nos moleste lo menos posible. Algunas soluciones más o menos típicas a este problema son:

* «LET L=USR n»: Si no empleamos la variable L para otra cosa, el resultado de «USR n» se almacena ahí, con lo que no interfiere con nuestros propósitos, aunque ocupa un espacio inútil en la zona de variables.

* «RAND USR n»: Esta es quizá la solución más empleada. Posiciona la serie de números pseudoaleatorios que se obtienen con la función «RND» del BASIC, según el contenido de BC al finalizar la rutina en CM. No ocupa ningún espacio extra, pero hay que tener en cuenta que si la rutina en CM se emplea como parte de un programa BASIC que utilice la función «RND» no obtendremos números aleatorios, porque cada vez que ejecutemos el CM posicionaremos la secuencia en el mismo sitio, consiguiendo siempre la misma serie de números «aleatorios».

* «LLIST USR n»: Si no tenemos previsto el uso de impresora con la rutina CM en cuestión (ni con el resto del programa), y hacemos que el resultado de «USR n» sea un número entre 0 y 9999 por el sencillo procedimiento de cargar BC con este número al final de la rutina, el ZX81 intentará listar por impresora después de ejecutar el CM, y al no estar ésta conectada, continuará normalmente. Lo mismo podría hacerse con «LPRINT USR n».

Al igual que con las otras funciones, se puede encadenar varias «USR», y por tanto varias rutinas en CM, en una misma línea de programa. Un ejemplo de esto podría ser «LET A= USR n + USR m + USR z», donde n, m y z, son las respectivas direcciones de memoria donde empiezan las distintas rutinas en CM.

Con «USR» puede llamarse tanto a rutinas alojadas en RAM como a las que existen en ROM, indistintamente, con lo que podemos aprovechar una serie de rutinas prefabricadas, que no son más que las que corresponden a las instrucciones BASIC, o a partes de ellas. En los Capítulos 19 y 20 se da amplia información sobre las más interesantes.

4.5. Los desastres

Mientras que el intérprete BASIC dispone de una serie de errores y detiene el programa cuando detecta alguno, dando información sobre el tipo de error y el número de línea en que se encuentra, en CM no disponemos de nada parecido.

El Z80 confía ciegamente en nosotros y hace todo lo que se le indica, por absurdo que sea. Además, una vez el ZX81 está trabajando en CM, ya no hay forma de detenerlo si no se ha previsto en la rutina.

Cuando por un error de programación obligamos al Z80 a hacer una barbaridad, el resultado suele ser lo que los ingleses llaman un «CRASH»: la pantalla acostumbra a quedar en blanco o con caracteres inesperados o deformados, y el ZX81 ya no obedece al teclado, siendo la única solución desconectarlo de la corriente y empezar otra vez desde cero.

Para que esto ocasione el mínimo de molestias es recomendable tomar una serie de precauciones, que conviene coger como hábitos:

—Nunca hay que correr un programa recién entrado o modificado sin haber realizado previamente una copia en cassette.

—Asegurarse en la medida de lo posible de que no se han cometido errores en la transcripción y entrada de los códigos, sobre todo en los que indican los saltos, así como que la secuencia del programa finaliza con una instrucción «RET», antes de probar a ejecutar la rutina en CM.

—Procurar estructurar los programas CM en bloques que puedan comprobarse a medida que se van entrando, para evitar tener que repasar páginas y páginas de listado en busca de un error que nos destroza el programa.

Es importante señalar que cualquier error que se cometa, por grande que sea, nunca puede perjudicar al ZX81, lo cual siempre es, por lo menos, un consuelo.

CAPÍTULO 5

LA INSTRUCCION «LD»

Comenzaremos el análisis detallado del juego de instrucciones del Z80 con una de las más potentes, a la vez que de las más empleadas. Está relacionada con asignar a una posición de memoria o a un registro el contenido de otra posición de memoria, registro o dato suministrado directamente; es decir, una especie de equivalente en CM del «LET» del BASIC. Tiene varios formatos que tienen en común las letras «LD», del inglés «LOAD», cargar. No tiene nada que ver con la instrucción «LOAD» del BASIC, porque aquí no cargamos un programa del cassete sino un byte desde otro sitio dentro de RAM, ROM, o del propio Z80.

En BASIC podemos hacer por ejemplo «LET A = B» cuyo análogo en CM sería «LD A,B», que copia el contenido del registro B en el registro A. Es una regla general de informática que *el acto de leer un registro nunca lo altera.*

Vamos a ver cada uno de los distintos formatos y su funcionamiento:

LD r1,r2: Copia «r2» en «r1» perdiéndose el contenido anterior de «r1». Cada una de estas combinaciones tiene un código distinto. (Ver Apéndice I).

«r1» y «r2» pueden ser: A, B, C, D, E, H, L.

Ejemplos: LD A,B y LD C,H. Son posibles todas las combinaciones.

LD r,n: Carga el registro especificado «r» con un dato fijo «n». El equivalente en BASIC sería «LET A = 118», por ejemplo. En CM la instrucción tendrá dos bytes: el primero depende del registro que queramos cargar y el segundo es el propio dato con el que cargaremos el registro. Así, para cargar el registro A con 118 dec. ($76), el mnemónico será «LD A,$76» con código 3E 76 hex.

«r» puede ser A, B, C, D, E, H, L.

LD A,(dd): Sólo existe con el registro A. «dd» es una dirección de memoria cuyo contenido se carga en A. Siempre que en un mnemónico encontramos algo entre paréntesis significa «el contenido de lo que hay en...». De esta forma, si en la posición de memoria $4074 hay $F5 y queremos cargar este dato en A, emplearemos «LD A,($4074)», que tendrá código 3A 74 40 hex. Nótese que, al construir el código, primero se coloca el byte menos significativo y después el más significativo, al revés de lo que parece más lógico. Esta es una norma de uso general para todas las instrucciones que emplean números de dos bytes o direcciones de memoria. El equivalente en BASIC de este formato de instrucción sería «LET A = PEEK dd».

LD (dd),A: Copia el contenido del registro A en la dirección de memoria «dd». El equivalente en BASIC sería «POKE dd,A».

LD A,(rr): «rr» puede ser HL,BC,DE. Carga en A el contenido de la dirección de memoria indicada por el par de registros «rr». Por ejemplo, si HL contiene $408F y en la dirección de memoria $408F hay $3D, al hacer «LD A,(HL)» cargamos $3D en A. Esto permite emplear un par de registros como puntero en un programa y traer al registro A el contenido de las posiciones de memoria que vaya punteando, como se verá con detalle más adelante.

«rr» puede ser también IX+d ó IY+d. En este caso se toma el contenido del registro IX ó IY (que son de 16 bits), se le suma «d» (un número entre $00 y $FF) sin poner el resultado en ninguna parte, y este resultado es la dirección de memoria cuyo contenido se carga en A. Es menos complicado de lo que parece. En el ZX81 el resgistro IY contiene $4000 (debe tener este valor al volver al BASIC) que corresponde al inicio de las variables del sistema. Si queremos cargar en A el contenido de la variable de sistema DF—SZ que está en la dirección 16418 dec. ($4022), por ejemplo, emplearemos «LD A,(IY+$22)» con código FD 7E 22 donde FD 7E es el código para la instrucción y $22 es el desplazamiento que hay que sumar al registro IY para que nos dé $4022, que es la dirección cuyo contenido queremos cargar en A.

LDr, (HL): «r» puede ser A,B,C,D,E,H,L. Funciona exactamente igual que el formato anterior, pero aunque ahora podemos emplear cualquier registro, el puntero deberá ser HL, (IX + d) ó (IY + d).

LD(rr),A: «rr» puede ser HL,DE,BC. Carga en la dirección de memoria indicada por el contenido del par «rr» el contenido de A. El equivalente en BASIC sería «POKE rr,A».

LD(HL), r: «r» puede ser A,B,C,D,E,H,L, ó un número entre $00 y $FF. Copia «r» en la dirección de memoria indicada por el contenido de HL, (IX+d), o (IY+d).

LD rr,nn: «rr» puede ser HL,DE,BC,IX,IY. «nn» es un número de dos bytes (entre $0000 y $FFFF) que se carga directamente en el par «rr». Al construir el código hay que recordar que primero va el código propio de la instrucción, luego el byte menos significativo y a continuación el byte más significativo de los que forman el dato «nn».

En el ZX81, el registro IX más vale no tocarlo para nada, porque interviene activamente en la rutina de DISPLAY, que irrumpe 50 veces cada segundo en nuestro propio programa. El contenido de IY se puede alterar pero, al regresar el control al BASIC, debe contener necesariamente $4000.

LDrr,(nn): «rr» puede ser HL,BC,DE,IX,IY. Carga en «rr» el contenido de la dirección de memoria «nn», que es un número de dos bytes cuyo código se coloca a continuación del de la propia instrucción de la forma ya explicada.

LD (nn),rr: «rr» puede ser HL,BC,DE,IX,IY y SP (el puntero de la pila [«stack»]. Carga en la dirección de memoria «nn» el contenido de «rr».

LD I,A y LD R,A: Se emplean muy raramente. Cargan en los registros especiales I (para interrupciones) y R (para refresco de RAM dinámica) el contenido de A.

NOTA: Los códigos de las instrucciones se encuentran en el Apéndice I. No se darán en las explicaciones de las instrucciones, aunque sí en los listados de los ejemplos. Una buena forma de aclarar puntos oscuros sobre el funcionamiento de una instrucción es buscarla en el listado de un ejemplo de los que se dan en este libro y ver qué hace en ese caso concreto.

CAPÍTULO 6

ARITMETICA SENCILLA: ADD, SUB, INC,DEC

El Z80 lleva incorporadas instrucciones para efectuar sumas y restas entre registros o entre parejas de registros. Al utilizarlas hay que tener en cuenta que si sobrepasamos la capacidad del registro que recibirá el resultado ($FF), éste sigue a partir de cero, de una forma análoga al cuentakilómetros de un automóvil cuando da la vuelta. Por tanto, cuando en una rutina una suma pueda sobrepasar en alguna ocasión el valor $FF, habrá que emplear una pareja de registros para efectuar la operación. De esta forma podemos obtener resultados hasta $FFFF (65535 dec.). El Z80 no puede manejar números mayores directamente; lo que hay que hacer entonces es organizar una estructura de varios bytes para almacenar el número y construir rutinas especiales para efectuar las operaciones. Un ejemplo de esto lo constituyen los números en coma flotante (en inglés «floating point»), que es como maneja la ROM del ZX81 los números cuando trabajamos en BASIC.

Para multiplicar o dividir números en CM hay que construir rutinas al efecto, excepto si queremos hacer HL = HL*DE, en cuyo caso podemos aprovechar una rutina que hay en la ROM del ZX81 a partir de la posición $1309, a la que llamaremos como una subrutina de nuestro programa.

Para operar directamente en CM tenemos cuatro instrucciones:

INSTRUCCION ADD: Trabaja con el registro A y otro cualquiera, sumando ambos y almacenando el resultado en A. El equivalente en BASIC sería «LET A= A + r» siendo r cualquier registro, de forma que el contenido previo de A se destruirá tras la suma. Así, para sumar el registro B al A utilizaremos la instrucción ADD A,B con código $80. Si en A tenemos $03 y en B $02, después de ejecutar la instrucción nos quedará A con $05 y B con $02.

También podemos sumar A directamente con un número, quedando el resultado almacenado en A. Se utilizará la instrucción «ADD A,n» siendo n un número entre $00 y $FF. Para ello colocaremos el código de la instrucción seguido del número que queremos sumar, de modo que la instrucción entera tendrá dos bytes. Para sumar $B6 al registro A, la instrucción será ADD A, $B6 con código $C6 B6. Al decodificar la instrucción el Z80 tomará el código siguiente al $C6 como el número que hay que sumar.

Con el formato «ADD A,(HL)» podemos sumar el byte indicado por el número contenido en el par de registros HL. El código de esta instrucción es $86. El equivalente en BASIC sería «LET A= A + PEEK HL».

De forma parecida trabajan «ADD A, (IX + d)» y «ADD A,(IY + d)», que emplean dos registros de 16 bits algo problemáticos en el ZX81: «d» es un número entre $00 y $FF que se suma al valor de IX ó IY, según la instrucción. El resultado de esta suma se toma como una dirección de memoria, y lo que contenga ésta se suma al registro A, almacenándose el resultado en A y quedando IX ó IY inalterados. Como en el ZX81 el registro IY tiene siempre $4000, se puede emplear ADD A,(IY+d) para leer las variables de sistema, que casualmente empiezan en $4000. Así, para sumar al registro A el contenido de la variable de sistema CDFLAG, que se encuentra en la posición $403B, la instrucción será «ADD A,(IY+$3B)» con código $FD863B, donde $FD86 corresponde a la instrucción en general y $3B es el desplazamiento que hay que sumar al registro IY para obtener $403B.

El Z80 ofrece además la posibilidad de sumar una pareja de registros cualquiera al par HL, tratándolos como si fueran registros de 16 bits. En este caso no tenemos la opción de sumar HL con un número de 16 bits especificado a continuación, como podíamos hacer con A y un número de 8 bits. El equivalente en BASIC de ADD HL,rr sería «LET HL= HL + rr» donde «rr» indica un par de registros.

INSTRUCCION INC: Significa «incrementar» y puede operar sobre un registro aislado o sobre una pareja, obteniendo como resultado el valor anterior del registro o pareja de registros más uno. El equivalente en BASIC sería «LET r= r + 1», que resultará familiar incluso para los más principiantes. Es una de las más empleadas en CM.

El formato «INC (HL)» incrementa en uno el contenido de la dirección de memoria indicada por el par HL directamente, sin tener que cargarlo previamente en ningún registro.

INSTRUCCION SUB: No hay un acuerdo acerca de si el mnemó-

nico es «SUB A,r» ó «SUB r» pero, en cualquier caso, lo que hace es restar el registro «r» de A almacenando el resultado en A, de forma parecida a como trabaja ADD. El equivalente en BASIC es «LET A = A — r». Se puede restar una cantidad fija de A con «SUB A,n» donde «n» es un número de 8 bits que se coloca tras el código de la instrucción ($D6) de modo análogo a «ADD A,n». Existen además «SUB A,(HL)», «SUB A,(IX+d)» y «SUB A,(IY+d)» que trabajan como sus correspondientes para la suma.

INSTRUCCION DEC: Significa «decrementar» y al igual que INC puede operar sobre un registro o una pareja. Decrementa en uno el valor del registro o pareja implicados. Su equivalente en BASIC es «LET r=r—1».

CAPITULO 7

LOS SEÑALIZADORES (FLAGS). INSTRUCCIONES RELACIONADAS CON FLAGS

Durante la ejecución de un programa hay una serie de condiciones que el Z80 debe conocer si se cumplen o no para operar correctamente, como por ejemplo si el resultado de una operación es cero, negativo (en complemento a dos), si se ha sobrepasado la capacidad del registro, etc. Para conocer esto la solución adoptada es una serie de indicadores que pueden estar a 1 ó a 0 según se cumpla o no la condición. Estos bits están agrupados en un registro especial F y constituyen los FLAGS. Se comportan como una bandera («Flag» es «bandera» en inglés) que sólo puede estar levantada (puesta a uno) si la condición se cumple, o bajada (puesta a cero) si la condición no se cumple.

Hay seis en total y algunos son de fácil acceso por programa, lo que resulta muy útil cuando necesitamos instrucciones del tipo «IF...THEN...» en CM, porque hay una serie de instrucciones que sólo se ejecutan cuando un flag determinado está a uno, por ejemplo.

Es muy importante comprender bien el funcionamiento de los flags y la forma de consultarlos para poder programar, en CM, aunque resulte poco familiar al principio debido a que en BASIC no se utiliza ningún tipo de flag. El sistema es más parecido al de la programación de las calculadoras de bolsillo. Sin embargo, utilizado correctamente es tan potente como el «IF...THEN...» del BASIC y con un poco de práctica es bastante cómodo de empleo. Hay que tener en cuenta que no todas las instrucciones alteran todos los flags. Hay una tabla detallada en el Apéndice II.

Primero analizaremos uno a uno con detalle, y luego veremos algunas instrucciones de las que los utilizan.

CARRY: En español acarreo, aunque no se emplea y suena bastante mal. Se coloca a uno cuando se sobrepasa la capacidad del registro o pareja de registros implicado en la instrucción, suponiendo que la instrucción en cuestión afecte a este flag. Es como si fuera un noveno bit del registro implicado en el resultado de la operación, y su funcionamiento corresponde al concepto «y me llevo una» cuando sumamos nueve más tres. Si al hacer esta suma sólo tuviésemos el lugar de las unidades para anotar el resultado, necesitaríamos un «carry flag» para indicar que 9 + 3 no son 2 sino 12. Por ejemplo si en el registro B tenemos $FF y ejecutamos INC B, el resultado será $00 en B y el flag CARRY puesto a uno. Si hacemos DEC B cuando B contiene $00 el resultado será $FF en B y el flag puesto a 1. Si B=$3E y hacemos INC B el resultado será B = $3F y el flag CARRY puesto a cero.

En el caso de que en la instrucción se opere con una pareja de registros, la situación es la misma pero esta vez el flag CARRY será una especie de decimoséptimo bit. El símbolo para representar CARRY es «C».

ZERO: Este flag indica cuándo el resultado de una operación efectuada en un registro o en una pareja es cero. Cuando el registro se pone a cero, el flag se pone a uno para indicarlo. Esto puede llevar a confusiones al principio, porque cuando el flag ZERO es cero quiere decir que el resultado *NO ES CERO,* al contrario de lo que intuitivamente se deduce a primera vista. El flag se llama /// ZERO pero se pone a UNO para indicar que la condición se cumple, es decir, que el resultado es cero.

Muchas de las instrucciones que trabajan con parejas de registros no alteran este flag mientras que sí lo hacen sus homólogos que trabajan con un registro, lo que puede llevar a confusiones al principio. Tal es el caso de ADD y DEC.

El símbolo para representar ZERO es «Z».

SIGN: Este flag indica el signo del resultado de la última instrucción que lo afecte, suponiendo que el número fuera interpretado con signo, lo que no tiene por qué ocurrir. En realidad es una copia del bit más significativo del registro implicado, es decir, del bit 7. Recordemos que los bits de un byte se numeran de derecha a izquierda 0, 1, 2, 3, 4, 5, 6, y 7.

Se utiliza en la comunicación de datos con el exterior a través

de bus y desde el punto de vista de la programación del ZX81 en CM en condiciones normales no reviste importancia, salvo en algunos saltos condicionales poco empleados generalmente.

El símbolo para representar SIGN es «S».

HALF—CARRY: Tiene muy escaso interés para el programador y, además, no es fácilmente accesible. Se comporta exactamente igual que CARRY, pero entre el cuarto y quinto bit, o sea entre los bits 3 y 4.

El símbolo para representar HALF—CARRY es «H».

ADD/SUBSTRACT: Tiene todavía menos interés que el anterior, entre otras cosas porque no es accesible directamente a través de ninguna instrucción, quedando limitado a su uso interno por el Z80. Además, para colmo se representa como «N». Se coloca a uno cuando la anterior operación ha sido una sustracción.

PARITY/OVERFLOW: El mismo flag tiene a su cargo dos funciones distintas.

Por una parte indica la «paridad» del resultado de la instrucción implicada. La paridad es un concepto que mide la cantidad de «1» de un byte, no importa dónde estén colocados. Según el total de unos sea par o impar, lo será la paridad del número. Así, 01011001 tendrá paridad «par» porque tiene cuatro unos y 4 es un número par. Cuando la paridad del resultado es «par», el flag PARITY/OVERFLOW se pone a 1 (que es curiosamente un número impar, así que préstese atención al principio) y cuando la paridad es «impar» el flag se pone a cero. Esto se utiliza normalmente para comprobar que no hay errores en la transcripción de datos al exterior pero, como resulta bastante evidente por su comportamiento, el ZX81 no lo emplea.

En inglés, «par» se escribe «even» y el símbolo de paridad par es «PE»; mientras que «impar» se escribe «odd» y el símbolo empleado es «PO».

El flag PARITY/OVERFLOW se comporta de esta forma en instrucciones lógicas, mientras que en instrucciones aritméticas es un indicador de OVERFLOW, esto es, de que un contenido del byte es sobreescrito accidentalmente durante la ejecución de la operación: durante la adición o sustracción de dos números en complemento a dos, el bit de signo (el bit 7) puede ser sobreescrito accidentalmente por el resultado y en este caso el flag PARITY/OVERFLOW se pone a uno.

Además, en las instrucciones de transferencia de bloques, que se verán con detalle más adelante, este flag se emplea para indicar que el contador BC ha llegado a cero, en cuyo caso el flag se coloca a «0». Lo mismo ocurre en las instrucciones de búsqueda en bloques.

7.1 Resumen de características de los flags

—Están agrupados en un registro especial cuyo símbolo es «F», ocupando un bit cada uno y dejando dos bits sin uso. El registro F está construido junto al A y se maneja junto a él en las instrucciones con parejas de registros que lo manipulan (PUSH, POP, etc.).

—Cada flag puede tener sólo dos estados: «1» cuando se cumple la condición que detecta, «0» cuando no se cumple.

—Cualquier flag queda en un estado indefinidamente hasta que se ejecuta una instrucción que lo altera.

—No todas las instrucciones afectan a los flags, aunque impliquen operaciones. Hay que consultar una tabla para saber si una instrucción altera un determinado flag.

—Normalmente, cuando una instrucción altera los falgs, su efecto se aprecia sólo sobre algunos de ellos.

—Hay una serie de instrucciones que se ejecutan únicamente si un determinado flag está en un estado dado. Este es el fundamento para construir en los programas los saltos condicionales y las sentencias del tipo «IF...THEN...» del BASIC, y de los bucles.

7.2. Algunas instrucciones relacionadas con los flags

Se analizarán aquí instrucciones de tres tipos:

1) Retornos de subrutinas, que se ejecutan como condicionales.

2) Operaciones aritméticas simples que tienen en cuenta al flag CARRY.

3) Comparaciones entre bytes como base para instrucciones condicionales.

7.3. Instruccion RET c (c = «condición»)

Ofrece la posibilidad de retornar de una subrutina sólo cuando se cumpla una condición determinada. Para saber si la condición se cumple, se explora el estado de un flag determinado. El concepto corresponde a la sentencia BASIC «IF condición THEN RETURN», que se comporta de modo parecido.

Para señalar en el mnemónico el estado del flag en el que la instrucción debe ejecutarse, y el propio flag que se va a explorar, se emplean unos símbolos que son los mismos para todas las instrucciones condicionales:

SIMBOLO	SIGNIFICADO	EJEMPLO
C	CARRY a «1»	RET C
NC	CARRY a «0»	RET NC
Z	ZERO a «1» (el resultado fue 0)	RET Z
NZ	ZERO a «0»	RET NZ
PE	PARIDAD «par»	RET PE
PO	PARIDAD «impar»	RET PO
M	SIGN a «1» (menos)	RET M
P	SIGN a «0» (más)	RET P

En todos los casos, si la condición se cumple se ejecuta un retorno al programa principal o al monitor BASIC, mientras que si no se cumple la instrucción se ignora y se pasa a ejecutar la siguiente.

INSTRUCCION ADC: Es un grupo de instrucciones para sumar datos, registros y parejas de registros que se comporta igual que ADD, con la única diferencia de que una vez efectuada la suma se añade el valor del carry. Es decir, realiza la suma teniendo en cuenta «las que llevamos» de la operación anterior. Esto nos permite trabajar con números mayores que 65535 ($FFFF), que por lo tanto ocuparán más de dos bytes cada uno.

Supongamos números de tres bytes, que nos permitirá una serie de valores entre $000000 a $FFFFFF (0 a 16777216 dec.). Con este formato no podremos efectuar operaciones llevando un número a cada registro o pareja de registros y ejecutando ADD o SUB. Habrá que operar el número en tres partes, una para cada byte de los que lo forman. Si tenemos un número en las posiciones $40BF, $40C0 y $40C1 y otro en $4382, $4383 y $4384 colocados de más a menos significativo, para sumarlos y dejar el resultado en $A1BB, $A1BC y $A1BD la rutina será la siguiente:

Vamos a emplear HL como puntero para el primer número, BC como puntero para el segundo y DE como puntero para el resultado:
Cargamos los punteros con las respectivas direcciones:

```
LD HL, $40C1
LD BC, $4384
LD DE, $A1BD
```

indicando los bytes menos significativos de cada número. A continuación cargamos en A el primer byte del segundo número y en B el primer byte del primero y los sumamos:

```
LD A,(BC)
LD B,(HL)
ADD A,B
```

Colocamos el resultado en el primer byte del resultado y decrementamos los punteros:

```
LD (DE),A
DEC HL
DEC BC
DEC DE
```

Suponiendo que la suma de los dos primeros bytes haya sido mayor que $FF, el flag CARRY se habrá puesto a «1»; si no estará a «0». Cargamos ahora en los registros los segundos bytes de los sumandos y efectuamos la suma, pero ahora con ADC, de forma que si hay carry lo sumaremos al resultado:

```
LD A,(BC)
LD B,(HL)
ADC A,B
```

Guardamos el resultado, decrementamos otra vez los punteros y volvemos a sumar los terceros bytes con ADC:

```
LD (DE),A
DEC HL
DEC BC
DEC DE
LD A,(BC)
LD B,(HL)
ADC A,B
```


Por último, guardamos el resultado y habremos completado la suma:

```
LD (DE),A.
```

INSTRUCCION SBC: Es el equivalente de ADC pero para restar. SBC A,r resta r de A y a continuación resta del resultado el flag CARRY. «r» puede ser un número, (HL), (IX+d), (IY+d), A, B, C, D, E, H, y L. Con el formato «SBC HL, rr» se efectúa la misma operación pero con parejas de registros, pudiendo ser «rr» HL, BC, DE o SP (SP es el puntero de la pila = «stack pointer»).
Como SUB no trabaja con parejas de registros, para hacer una resta con dos números de 16 bits se emplea «SBC HL,rr» asegurándose previamente que el flag CARRY está a cero. Una de las instrucciones que coloca C a cero es «AND A», como se verá más adelante. Por tanto:

```
AND A
SBC HL, DE
```

es lo mismo que «SUB HL,DE», si existiera esta instrucción.

INSTRUCCION SCF: Son las siglas de «Set Carry Flag». Tiene como único efecto poner a uno el flag CARRY.

INSTRUCCION CP: Tiene la forma «CP r» donde «r» puede ser (HL), (IX + d), (IY + d). A, B, C, D, E, H, L ó un número. Se emplea para comparar «r» con A y opera de la forma siguiente: efectúa A−r pero el resultado no se guarda en ningún sitio, por lo que todos los registros quedan inalterados. Lo que hace es posicionar todos los flags de acuerdo con el resultado obtenido, con lo que nos quedan preparados para emplear a continuación instrucciones condicionales.

Si A es mayor que «r» entonces C es cero
Si A es igual que «r» entonces C y Z son cero
Si A es menor que «r» entonces C es uno.

Por ejemplo CP r seguido de RET Z sería en CM el equivalente de «IF A=r THEN RETURN» del BASIC. En la forma «CP A» se posicionan los flags tras efectuar A−A, es decir, todos los flags quedan a cero.

INSTRUCCION CCF: Siglas de «Complement Carry Flag». No afecta a ningún registro y complementa el flag CARRY, poniéndolo a cero si estaba a uno y viceversa. El formato es «CCF».

EJERCICIO

Construir una rutina en CM que calcule la cantidad de memoria libre para el usuario en un momento determinado, expresada en bytes; es decir, la memoria disponible hasta llenar el ZX81. Prescindir de los bytes ocupados por la pila («stack») del Z80.

PISTA 1: Hay que calcular la longitud de la zona de memoria libre (ver apartado 2.3), que se encuentra entre la «pila («stack») del calculador» y la «pila («stack») de gosub».

PISTA 2: La zona buscada se encuentra acotada por dos variables de sistema: STKEND (direcciones $401C y $401D) y EERSP (direcciones $4002 y $4003).

PISTA 3: La resta del contenido de estas dos variables de sistema nos dará la cantidad de memoria libre usuario buscada.

PISTA 4: Un sistema puede ser: Cargar HL con el contenido de la variable de sistema ERRSP y DE con el contenido de STKEND, restarlos y trasladar el resultado al par BC, para que al ejecutar la rutina con «PRINT USR» nos dé en pantalla el resultado. (Ver apartado 4.4).

SOLUCION: Para restar HL — DE emplearemos «SBC HL,DE» habiendo puesto previamente a cero el flag CARRY por el sistema de ponerlo a uno con «SCF» y luego complementarlo con «CCF»:

```
2A 02 40      LD HL,($4002)
ED 5B 1C40    LD DE,($401C)
37            SCF
3F            CCF
ED 52         SBC HL,DE
44            LD B,H
4D            LD C,L
C9            RET
```

Si colocamos la rutina en una línea 1 REM con 14 espacios, su dirección inicial será 16514. Con «PRINT USR 16514» obtendremos en pantalla la cantidad de memoria libre usuario. Hay que utilizar el cargador hexadecimal del apartado 2.7 para entrar la rutina. Son posibles otras soluciones igualmente válidas.

CAPITULO 8

SALTOS: ABSOLUTOS, RELATIVOS Y CONDICIONALES

Un programa en CM se ejecuta byte a byte, por orden de menor a mayor hasta que se encuentra un RET que devuelve el control al monitor BASIC. Resultaría muy poco útil programar en CM si no pudiésemos alterar esta secuencia normal de ejecución y para hacer una cosa 50 veces tuviésemos que ponerla otras tantas una a continuación de otra. Sería como si en BASIC no existiera «GOTO» ni «FOR...NEXT». Afortunadamente hay una serie de instrucciones en CM para efectuar saltos dentro del programa, tanto condicionales como incondicionales. En cada uno de estos dos tipos podemos realizar saltos absolutos o relativos.



saltos absolutos

SALTOS ABSOLUTOS: Cuando en BASIC queremos saltar a otra parte del programa disponemos de «GOTO n», donde «n» es la línea de programa a la que queremos transferir el curso del proceso.

En CM las instrucciones no se identifican por un número de línea sino por la posición de memoria que ocupan. La instrucción homóloga a «GOTO» es JUMP (en español «salto»), con mnemónico «JP nn» donde «nn» es un número de 16 bits que indica la posición de memoria a la que se efectúa el salto. Así, para saltar a una instrucción situada en la posición de memoria $4093 emplearemos «JP $4093». Al efectuar un salto hay que tener en cuenta que si por error el programa «cae» en medio de una instrucción de dos o tres bytes, el Z80 interpretará a partir de ahí todos los códigos desplazados una posición, lo que no tendrá sentido y ocasionará un desastre de los típicos en CM.

La forma «JP (HL)» saltará a la posición de memoria indicada por el par de registros HL. El equivalente en BASIC es «GOTO HL». Esto sólo puede hacerse con las parejas HL, IX e IY.



saltos relativos

SALTOS RELATIVOS: Otra forma de hacer saltos en CM que no existe en BASIC, se basa en el concepto «saltar n bytes hacia adelante o hacia atrás» en lugar de «saltar a la posición nn», como en «JP nn». La instrucción es JUMP RELATIVE (salto relativo) con menmónico «JR e» donde «e» es el número de bytes que hay que saltar hacia adelante o hacia atrás. Como el salto puede ser en dos sentidos, el Z80 interpreta en complemento a dos el byte «e» (ver apartado 1.1). Como consecuencia sólo podremos saltar entre 127 bytes hacia adelante y 128 hacia atrás. Para calcular el valor del byte del salto en complemento a dos puede ser útil la tabla del Apéndice III.

El Z80 suma automáticamente dos al valor del salto antes de efectuarlo. De esta forma, como la instrucción «JR e» es de dos bytes, «JR $00» no tiene ningún efecto, siguiendo el programa normalmente, mientras que «JR $FE» (—2 en notación de complemento a

dos) vuelve a ejecutar la instrucción entrando en un lazo infinito del que no hay forma de salir. Para calcular el valor de un salto relativo en un listado lo mejor es empezar a contar a partir del primer byte de la instrucción que sigue a «JR e», tanto para saltos hacia adelante como hacia atrás, y luego buscar el valor en la tabla.

Otra ventaja de los saltos relativos es que funcionan independientemente de las posiciones de memoria donde se encuentre el programa, con lo que si cambiamos su localización seguirá funcionando perfectamente. Además mientras que los saltos absolutos ocupan tres bytes, los relativos sólo ocupan dos. Como normal general siempre que se pueda es mejor emplear saltos relativos.

SALTOS CONDICIONALES: Constituye un grupo de instrucciones que da una gran potencia a la programación en CM. Junto con las instrucciones que afectan a los flags permiten construir sentencias equivalentes a «IF condición THEN GOTO» del BASIC. Pueden ser saltos absolutos o relativos. Solamente se ejecutan si un determinado flag está en un estado específico, y si no la instrucción es ignorada y se continúa normalmente el programa.

Por ejemplo «JP NC nn» salta a la posición de memoria «nn» en el caso de que el flag C esté a cero. Los mnemónicos se construyen teniendo en cuenta los símbolos del Capítulo 7. Algunas de las combinaciones que más se utilizan son las siguientes:

INSTRUCCIONES	EQUIVALENTE EN BASIC
CP r JP Z nn	IF A=r THEN GOTO nn
CP r JP NZ nn	IF A<>r THEN GOTO nn
CP r JP C nn	IF A<r THEN GOTO nn
CP r JP NC nn	IF A> = r THEN GOTO nn
CP $00 JP M nn	IF A> = 128 THEN GOTO nn
CP $00 JP P nn	IF A< 128 THEN GOTO nn

Este tipo de instrucciones se emplea además para construir lazos y bucles, como se verá en el Capítulo 11.

CAPÍTULO 9

LA PILA («STACK»)

En el transcurso de un programa se cumple (o debería cumplirse) aquello de «un lugar para cada cosa y cada cosa en su lugar». Sin embargo hay veces en que un dato nos estorba momentáneamente en un registro y no merece la pena guardarlo en su lugar de la memoria sólo para tenerlo que traer al cabo de unas pocas instrucciones. En estos casos resulta muy útil una estructura en la que «apilamos» literalmente los datos que nos estorban, y que volveremos a recuperar tan pronto como los necesitemos de nuevo.

Esta estructura es la PILA, que también se expresa con su equivalente inglés «STACK». Imaginemos una pila de cajas de zapatos en las que podemos guardar números. La única caja a la que tendremos acceso será la que se encuentra en la cima de la pila, y cuando guardemos una nueva caja lo tendremos que hacer sobre ésta. Supongamos además que tenemos un ayudante que nos trae y traslada las cosas a las cajas, con lo que no tenemos que preocuparnos de dónde está realmente la pila. Para localizar el contenido de una caja lo único que necesitamos saber es el número de cajas que tiene por encima, y decir al ayudante que vaya trayendo cajas hasta que la que buscamos sea accesible. Si hacemos una prueba con objetos (cassettes o libros apilados) nos daremos cuenta que recogemos las cosas en orden inverso a como las apilamos.

La pila («stack») del Z80 reúne todas estas características, aunque hay una pequeña diferencia con el ejemplo de las cajas que puede resultar un tanto sorprendente: la pila («stack») se encuentra «colgada» del techo en lugar de estar apoyada en el suelo como una pila convencional: tiene la cima en posiciones de memoria inferiores

a las de la base y crece hacia abajo. Esto es así al parecer por razones técnicas y de eficacia, y una vez hechos a la idea, la comodidad para nosotros es la misma.

En el Z80 sólo es posible guardar en la pila o «stack» parejas de registros, nunca un registro aislado, lo que no presenta mayores problemas. Cuando nos interesa sólo guardar un registro lo que hacemos es guardar la pareja entera, teniendo en cuenta que cuando los recuperemos destruiremos el contenido anterior de los dos registros, tanto del que nos interesa como de su pareja. Puesto que sólo podemos trabajar con parejas de registros, cada vez que guardemos algo en la pila («stack»), ésta crecerá en dos bytes (hacia abajo) y al recuperarlo se reducirá en otros dos bytes.

El Z80 guarda la dirección de la cima de la pila («stack») en un registro especial de 16 bits: el registro SP («Stack pointer = puntero de pila»), que él mismo se actualiza automáticamente cuando es necesario. Para utilizar la pila («stack») disponemos de dos instrucciones: PUSH y POP.

INSTRUCCION PUSH: Tiene la forma «PUSH rr» donde «rr» puede ser BC, DE, HL, AF, IX ó IY. Se emplea para guardar el contenido de la pareja de registros especificada en la pila («stack»). En realidad, lo que hace es copiar el contenido del primer registro en la dirección indicada por el registro SP−1, el contenido del segundo registro en la posición SP−2, y decrementa el puntero SP haciendo SP=SP−2, todo ello de forma automática. El contenido de la pareja «rr» no se altera por el hecho de guardarlo en la pila y se puede seguir utilizando.

INSTRUCCION POP: El formato es «POP rr», donde «rr» significa lo mismo que en PUSH y su efecto es asignar a la pareja de registros «rr» los dos bytes de la cima de la pila («stack») destruyendo lo que tengan en ese momento. En realidad se copia el contenido de la dirección SP en el segundo registro especificado, el contenido de la dirección SP−1 en el primer registro, y se hace SP=SP+2. En teoría, si decrementásemos el registro SP en dos unidades volveríamos a tener la pila igual que antes del POP. Lo que ocurre es que en el ZX81 la rutina de DISPLAY utiliza también la pila («stack») para sus propósitos. Como actúa por su cuenta unas 50 veces por segundo, corremos el riesgo de que antes de restablecer «a mano» el puntero SP la rutina de DISPLAY nos haya cambiado lo que tenemos sobre la cima de la pila («stack»). De todas formas utilizando sólo las instrucciones PUSH y POP no tendremos ningún problema.

El par AF está formado por el registro A y el F, que contiene los flags. A efectos de PUSH y POP se comportan como otra pareja de registros cualquiera. PUSH AF se suele emplear para guardar el registro A dejándolo libre para operaciones que no se pueden realizar sobre ningún otro, como ADD A,n ó LD A,(DE).

Por ejemplo, para hacer «LD C,(DE)», que no existe como tal instrucción, sin alterar ningún otro registro, se puede emplear:

```
PUSH AF     Guarda A en la pila («stack»)
LD A,(DE)   Carga A con (DE)
LD C,A      Carga C con el contenido de A
POP AF      Recupera A de la pila («stack»)
```

Otro empleo típico consiste en guardar los registros cuando llamamos a una subrutina que destruye el contenido previo de los registros y nos interesa seguir trabajando con ellos:

```
PUSH AF
PUSH HL     Guarda los registros en la pila («stack»)
PUSH BC
PUSH DE
CALL nn     Llama a la subrutina
POP DE
POP BC      Recupera el contenido de los registros de la
            pila («stack»)
POP HL
POP AF
```

La pila («stack») resulta muy útil para construir bucles anidados, como se verá en el Capítulo 11.

El puntero de la pila se puede alterar con las siguientes instrucciones:

```
LD SP,HL
LD SP, nn
LD SP,(nn)
LD SP,IX
LD SP,IY
ADD HL,SP
ADC HL,SP
SBC HL,SP
ADD IX,SP
ADD IY,SP
INC SP
DEC SP
```


"H"	puntero aquí tras POP HL.
"L"	
"H"	puntero aquí con INC SP
"L"	puntero SP indicando esta dirección

Figura 3

Podemos intercambiar los contenidos de dos pares de registros sin alterar ningún otro haciendo:

```
PUSH HL
PUSH DE
POP HL
POP DE
```

También podemos intercambiar por ejemplo el contenido de H y L, sin alterar otros registros:

```
PUSH HL
PUSH HL
INC SP
POP HL
INC SP
```

Esta operación se entenderá mejor si tenemos en cuenta que tras las dos primeras instrucciones la pila queda como se ve en la figura 3.

Al decrementar SP en uno recuperamos con POP un byte de cada PUSH y luego hay que restablecer la situación con INC SP para que no nos quede el contenido de H en la pila («stack»).

CAPÍTULO 10

TRABAJANDO CON BITS

Aunque en muchas ocasiones podemos prescindir del formato binario de los números recordemos que en realidad es en forma de unos y ceròs como trabaja el Z80. Hay situaciones en las que considerando los datos en binario tenemos posibilidades que se nos escapan en Hex, o decimal. En CM disponemos de una serie de instrucciones para alterar o explorar un solo bit determinado de un registro o dirección de memoria, efectuar operaciones lógicas o rotaciones de los bits dentro del byte. Vamos a estudiar cada uno de estos grupos, teniendo en cuenta que en un byte los bits se numeran de izquierda a derecha: 7, 6, 5, 4, 3, 2, 1, 0.

INSTRUCCION SET: Tiene la forma «SET n,r» donde «n» es el número del bit (de 0 a 7) y «r» puede ser A, B, C, D, E, H, L (HL), (IX+d) o (IY+d). El efecto es poner a «1» el bit «n» del registro o posición de memoria especificados. Así si en B tenemos $00 y hacemos SET 3,B nos quedará 00001000 en B.

INSTRUCCION RES: Con el formato «RES n,r», tiene el efecto contrario a SET. RES es una abreviatura de RESET y pone el bit «n» del registro «r» a cero.

INSTRUCCION BIT: El formato es «BIT n,r» donde «n» es el número de bit y «r» tiene el mismo significado que en SET o RES. El efecto de esta instruccion es explorar el bit «n» del registro o dirección «r» colocando el flag ZERO según el estado de ese bit. No

altera el contenido de ningún registro, es decir, el bit explorado no se modifica. Si el bit explorado es «1» se pone Z a «1» y si es «0» se pone Z a «0» lo que permite ejecutar saltos condicionales en función del estado del bit explorado. Por ejemplo:

```
CB 72        BIT 6,D
C3n2n1       JP Z,nn
```

saltará a la dirección «nn» si el bit 6 del registro D es un cero. Con BIT 0,A podemos saber si el contenido de A es par (dará Z=0) o impar (dará Z=1).

INSTRUCCION CPL: Sólo tiene la forma «CPL» y trabaja únicamente sobre el registro A, llamado también acumulador. El efecto es realizar el complemento a uno de A; es decir, cambia cualquier uno por cero y cualquier cero por uno. Por ejemplo si en A tenemos 01100010 y ejecutamos CPL obtenemos 10011101. No se alteran los flags por el uso de CPL. Ejecutar CPL dos veces consecutivas es como si no hubiéramos hecho nada.

INSTRUCCION NEG: Opera sólo sobre A realizando un complemento a dos, es decir, ejecutando 0—A, sin alterar ningún otro registro. El equivalente en BASIC sería «LET A= —A». Por ejemplo, si hacemos:

```
ED 44        NEG
86           ADD A,(HL)
```

obtendremos en A el resultado de restar el contenido de la dirección (HL) menos A, o sea (HL)—A.

INSTRUCCION DAA: También trabaja únicamente sobre A y significa «Decimal Adjust Acumulator», ajustando A a decimal. Realiza la conversión a decimal del valor de A considerando que la operación anterior (suma o resta) ha sido realizada con números decimales en lugar de hexadecimales. Se verá más claro con un ejemplo: Si queremos sumar 13 y 27 dec. sin tener que pasarlos a Hex., podemos hacer LD A, 13 y ADD A,27 obteniendo en A $3A que es el resultado de sumar $13 y $27, tal como trabaja normalmente el Z80. Pero si ahora hacemos DAA obtendremos en A 40, que es el resultado de sumar *en decimal* 13 y 27. DAA trabaja bien tanto si la operación fue una suma como si fue una resta.

INSTRUCCION OR: La forma es «OR r» donde «r» puede ser (HL), (IX+d), (IY+d), A, B, C, D, E, H, L, o un número. Realiza un «O» lógico entre A y el registro o dato especificado, guardando el resultado en A. Esta operación se realiza comparando bit a bit cada uno de los de A con el que ocupa la misma posición en el registro o dato de comparación. Si ambos son «0» el resultado es «0»; si uno de los dos o ambos están a «1» el resultado es «1», es decir:

ACUMULADOR	REGISTRO O DATO	RESULTADO EN A
0	0	0
0	1	1
1	0	1
1	1	1

Por ejemplo, si A contiene 10011011 y C contiene 00110110 al hacer «OR C» obtenemos:

A	10011011
C	00110110
RESULTADO A	10111111

«OR A» deja A inalterado pero pone el flag C (CARRY) a cero. Lo mismo hace «OR $00» aunque ésta es una instrucción de dos bytes.

INSTRUCCION AND: Con formato «AND r» donde «r» significa lo mismo que para OR, realiza un «Y» lógico entre el acumulador (A) y el registro o dato especificado, comparando bit a bit y dando como resultado «0» cuando uno de los bits es «0» y «1» cuando ambos son «1»:

ACUMULADOR	REGISTRO O DATO	RESULTADO EN A
0	0	0
1	0	0
0	1	0
1	1	1

En el ejemplo de la instrucción OR haciendo AND C obtenemos:

A	10011011
C	00110110
RESULTADO A	00010010

«AND a» deja A inalterado pero pone el flag C (CARRY) a cero. «AND 00001111» deja en A solamente los cuatro primeros bits, que corresponden al dígito menos significativo del dato en Hex. Por ejemplo, si tenemos $CD en A y hacemos «AND $0F» ($0F = 00001111 bin.) obtenemos en A $0D.

INSTRUCCION XOR; Con el mismo formato que las anteriores, realiza un «O» exclusivo. El resultado se obtiene como en «OR r» pero si ambos bits son «1» el resultado es «0»:

ACUMULADOR	REGISTRO O DATO	RESULTADO EN A
0	0	0
1	0	1
0	1	1
1	1	0

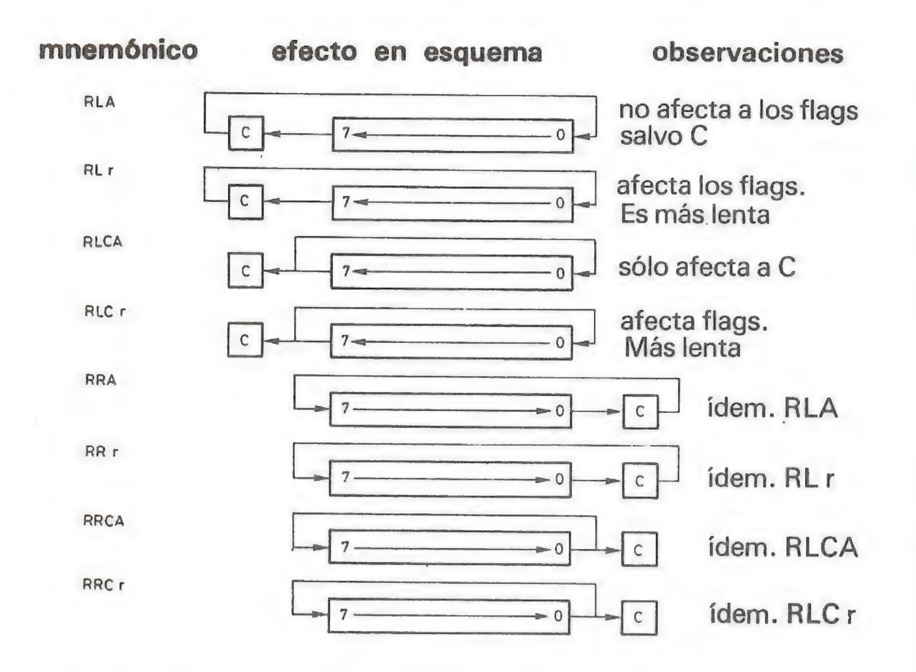
En el ejemplo anterior haciendo «XOR C» obtenemos:

A	10011011
C	00110110
RESULTADO A	10101101

«XOR A» tiene como efecto poner el flag C (CARRY) a cero y A también a cero, todo con una sola instrucción.

DESPLAZAMIENTOS Y ROTACIONES: La serie de instrucciones que veremos a continuación tienen en común que los bits se desplazan dentro del byte pasando a ocupar cada uno la posición de su vecino. Esto puede hacerse hacia la derecha o hacia la izquierda, y las diferencias estriban en los bits que entran por el extremo desocupado y en el destino del bit «expulsado» fuera del byte por el otro extremo.

ROTACIONES: El bit «expulsado» del byte se copia en el flag CARRY (C), y el contenido previo de éste «entra» por el extremo que ha quedado vacío. El significado de «r» en la siguiente tabla puede ser: (HL), (IX+d), (IY+d), A, B, C, D, E, H, y L.



En las instrucciones RLC y RRC entra por el extremo libre el bit que sale por el otro, que al mismo tiempo se copia en C.

Hay además dos instrucciones un tanto raras y muy poco usadas:

RLD A 7 4 3 0 7 4 3 0 (HL)

RRD A 7 4 3 0 7 4 3 0 (HL)

Estas rotaciones se efectúan en bloques de cuatro bits entre el registro A y la dirección de memoria indicada por HL.

—**Unos ejemplos:**

Contenido de A: 10010110
Estado de C: 0

Operación	**Resultado**
RLA	A: 00101100 y C:1
RRA	A: 01001011 y C:0
RLCA	A: 00101101 y C:1
RRCA	A: 01001011 y C:0

DESPLAZAMIENTOS: El bit «expulsado» va a parar al flag C, el contenido previo de éste se pierde y por el extremo libre entra o bien un cero o bien una copia del bit 7. El significado de «r» es el mismo que para las rotaciones.

Mnemónico	**EFECTO en esquema**
SLA r	C ← 7 ← 0 ← 0
SRA r	↳7 → 0 → C
SRL r	0 → 7 → 0 → C

—**Más ejemplos:**

Contenido de A:10010110
Estado de C: 0

Operación	**Resultado**
SLA A	A: 00101100 y C:1
SRA A	A: 11001011 y C:0
SRL A	A: 01001011 y C:0

CAPÍTULO 11

BUCLES Y SUBRUTINAS

Todo lo relativo a subrutinas en CM es prácticamente igual que la forma de tratarlas en BASIC y su comprensión no presentará problemas. La instrucción equivalente a «RETURN» es en CM «RET», como se vio en el apartado 4.3. El equivalente a «GOSUB» es «CALL nn» donde «nn» es una dirección de memoria a la que se transfiere el control del programa de forma idéntica a como en «GOSUB» se transfiere a un número de línea. Al ejecutar «CALL nn» se guarda automáticamente en la pila («stack») la dirección de la siguiente instrucción que debería ser ejecutada si no existiera la llamada a la subrutina (la siguiente instrucción del listado) y se carga el registro PC (contador de programa) con la dirección «nn» de CALL, con lo que la siguiente instrucción que se ejecutará será la que se encuentra en la dirección «nn». Con el formato «CALL c,nn» (c = condición) podemos hacer llamadas condicionales a subrutinas de la misma forma que en los saltos condicionales (ver Cap. 8). Por ejemplo «CALL Z,$4082» llamará a una subrutina que empiece en la dirección $4082 sólo en el caso de que el flag Z esté a cero. El equivalente BASIC de este tipo de instrucciones sería «IF condición THEN GOSUB...».

Mientras que el BASIC tiene una pila o «stack» especial para almacenar las líneas de retorno de subrutinas (la «pila "stack" de GOSUB», ver apartado 2.3) en CM se emplea la propia pila («stack») del Z80 para ello. Por tanto la instrucción RET es una especie de «POP PC», siendo PC el registro especial de 16 bits que utiliza el Z80 como contador de programa. Esto quiere decir que al ejecutar

un RET el programa regresa a la dirección de memoria que se encuentra almacenada en la cima de la pila («stack») quede en su situapués de ejecutar una rutina la pila debe quedar igual que antes de llamarla. Hay que tener pues cuidado de que cuando se empleen PUSH o POPs en una subrutina la pila («stack») quede en su situación original. Por ejemplo el programa:

dirección	**mnemónico**
$4082	CALL $408A
	•
	•
$408A	PUSH HL
	RET

no funcionará porque al ejecutar RET la dirección que se colocará en PC será el contenido del par HL y por tanto será ahí a donde saltará el programa.

Esto se puede emplear para hacer «volver» una subrutina a una dirección diferente a la de partida, que puede ser distinta según se cumplan determinadas condiciones en el transcurso de la propia subrutina. Para ello no tenemos más que traer la antigua dirección de retorno de la pila («stack») con un POP y colocar la nueva con un PUSH. Por ejemplo, para que una subrutina «regrese» a la dirección $4000 haremos:

E1	POP HL ... recuperamos de la pila la dirección original de retorno
210040	LD HL,$4000
E5	PUSH HL ... colocamos en la pila la nueva dirección
C9	RET

Si en una subrutina hemos hecho algunos PUSH y en un momento dado queremos efectuar un RET a la dirección original, un recurso es almacenar la posición del puntero de la pila («stack pointer») SP en alguna parte de la memoria con «LD (nn),SP» y antes de ejecutar RET devolver el puntero a su posición original con «LD SP,(nn)» sin tener que preocuparnos de restablecer la situación original en la pila.

Todas las formas de CALL son de direccionamiento absoluto, es decir, no tenemos la posibilidad de llamadas relativas análogas a los saltos relativos «JR e» (ver Capítulo 8).



11.1. Estructuras de bucles en CM

En BASIC hay dos tipos fundamentales de bucles: Uno es el que se forma definiendo un contador, por ejemplo «LET I=0» en una línea «nn» y al final del proceso del bucle incrementar el valor del contador con «LET I=I+1» y hacer un salto condicional si el contador ha alcanzado el valor límite con «IF < I = 5 THEN GOTO nn + 1».

Otra estructura de bucle viene dada por el conjunto de las instrucciones «FOR I=0 TO 5» y «NEXT N», que tendrán el mismo efecto que antes pero haciéndolo de una forma más automática.

En CM podemos construir ambos tipos de bucles utilizando prácticamente los mismos conceptos pero, naturalmente, con instrucciones diferentes.

1) Tipo contador-test condición-salto condicional: Emplearemos un registro cualquiera (que no vayamos a utilizar para otra cosa) como contador. A continuación colocaremos el contenido del bucle, y al final efectuaremos el test con la instrucción o serie de instrucciones adecuadas. Entonces emplearemos una instrucción de salto condicional de acuerdo al test efectuado, que realice el salto al principio del bucle. Supongamos que vamos a ejecutar un bucle cinco veces, empleando el registro C como contador:

```
0E 05       LD C,5   inicializamos el contador
            $5555 dirección del inicio del bucle
                 •
                 •
                 •        contenido del bucle
                 •
45          DEC C decrementamos el contador. Esta instrucción
                  pone el flag Z a uno cuando el contenido es
                  cero
```



```
C25555      JP NZ $5555 salto si C no es cero. Se repetirá 5 veces
```

Se puede emplear tanto saltos absolutos como relativos, y cualquier tipo de instrucción de test, como «CP r». Si utilizamos un registro único como contador sólo podremos repetir el bucle un máximo de 256 veces. Para un número mayor de repeticiones tendremos que emplear un par de registros. En este caso habrá que hacer un test explícito para comprobar si hemos llegado a cero, porque la instrucción «DEC rr» no altera los flags. Una solución es:

```
0B          DEC BC
78          LD A,B
B1          OR C
C2 n2n1     JP NZ inicio
```

El resultado de hacer un «OR» lógico entre los contenidos de B y C será cero sólo en el caso de que tanto B como C sean cero (ver Cap. 10).

Las soluciones propuestas hasta ahora nos ocupan algún registro como contador que podemos necesitar para otras cosas en el interior del bucle. Una forma de realizar bucles manteniendo todos los registros libres en su interior, es empleando la pila («stack») para almacenar el contenido del contador mientras se ejecuta el bucle:

```
01 11 11    LD BC,$1111
C5 $4000    PUSH BC   guardamos el contador en la pila («stack»)
              •
              •         contenido del bucle
              •
C1          POP BC    recuperamos el contador
78          LD A,B
B1          OR C      comprobamos si es cero
C2          JP NZ,$4000 salta al inicio del bucle, donde lo vol-
                        vemos a guardar
```

2) Tipo contador—test y salto condicional automáticos: Hay una instrucción en CM que se parece un poco a «FOR...NEXT»: es «DJNZ e». Emplea siempre como contador el registro B y tiene el efecto de decrementar B en una unidad, mirar si es cero, y si es cero hacer un salto condicional relativo como «JR NZ,e». El primer ejemplo de bucle que hemos visto quedaría así empleando DJNZ e:

```
         LD B,5   el contador es siempre B
i n i c i o  •
         •        contenido del bucle
         •
         DJNZ e
```

donde «e» será el número de bytes hacia atrás que debe saltar para caer en la dirección de memoria marcada como «inicio» (ver Cap. 8 para este tipo de saltos relativos).

Aunque «DJNZ e» es más cómodo de usar, tiene el inconveniente de emplear siempre el registro B como contador, y además sólo se puede construir bucles de 127 bytes de longitud como máximo, al utilizar saltos relativos.

Algunas cosas a tener en cuenta al emplear DJNZ:

—Los saltos pueden ser tanto hacia adelante como hacia atrás, o saltar la propia instrucción con «DJNZ $FE», que es una forma de perder tiempo.

—Primero se decrementa B en uno y luego se efectúa el salto condicional, luego si B = 0 el resultado será B = $FF y el salto se ejecutará.

—Se pueden construir bucles anidados si empleamos la pila («stack») para almacenar los respectivos valores de B:

```
         LD B,$05
LAZO 1   PUSH BC
         •
         •
         •
         LD B,$04
LAZO 2   PUSH BC
         •
         •
         LD B,$02
LAZO 3   PUSH BC
         •
         •
         POP BC
         DJNZ LAZO 3
         POP BC
         DJNZ LAZO 2
         POP BC
         DJNZ LAZO 1
```


tiene la misma estructura y comportamiento que

```
FOR N = 1 TO 5
•
•
FOR M = 1 TO 4
•
•
FOR P = 1 TO 2
•
•
NEXT P
NEXT M
NEXT N
```

CAPÍTULO 12

INSTRUCCIONES CON REPETICION

Cuando en CM realizamos una tarea trabajando dentro de un bloque, es decir, dentro de un conjunto de bytes correlativos con unas características comunes, se suelen emplear dos elementos típicos para gestionarlo:

—Un registro o par de registros como puntero dentro del bloque, que contienen la dirección de memoria del byte dentro del bloque sobre el que estamos trabajando.
—Un registro o par de registros como contador, que nos indicarán cuándo hemos llegado al final del bloque, suponiendo que tenga una longitud fija conocida.

Como es lógico, utilizaremos registros aislados cuando la longitud sea inferior a 256 bytes, o un par de registros si es mayor.
El Z80 dispone de una serie de instrucciones que automatizan en mayor o menor medida parte de la gestión de estos dos elementos, lo que facilita la tarea de programación en estos casos.
Supongamos que tenemos un bloque de 500 bytes en el que queremos localizar un byte que contiene $AA para saber su posición dentro del bloque. Una forma de localizarlo será construir un bucle que vaya explorando cada byte y realizando una comparación hasta encontrar el buscado. Si el bloque comienza a partir de la dirección $4000, y empleamos el par BC como contador, el registro A para traer y comparar cada byte, y el par HL como puntero dentro del bloque, el listado queda:


```
INICIO    LD HL,$4000          inicializamos el puntero
          LD BC,500 dec.       y el contador
OTRO      LD A,$AA             cargamos en A el número a
                               buscar
          CP (HL)              compara A con el contenido
                               de (HL)
          JR Z, ENCONTRADO     si son iguales está localizado
          INC HL               siguiente byte del bloque
          DEC BC               decrementa el contador
          LD A,B               prueba si BC ha llegado a cero
          OR C                 (ver Cap. 12)
          JR NZ, OTRO          repite para el siguiente byte
NO ESTA   . . . . . . .        si llega aquí, el bloque no
                               contiene $AA
```

Esta rutina queda muy simplificada por el empleo de la instrucción «CPI», que de forma automática realiza por sí sola CP(HL), INC HL, y DEC BC (CPI son las siglas de Comparar con Incremento) cada vez que se ejecuta. Por tanto la rutina queda simplificada de forma:

```
INICIO    LD HL,$4000
          LD BC,500 dec.
          LD A,$AA
OTRO      CPI
          JR Z, ENCONTRADO
          JP PE, OTRO
NO ESTA   . . . . . . .
```

Se puede apreciar que es bastante más corta. Para detectar si BC es igual a cero, la instrucción pone el flag P/V a cero cuando BC=0 ó a uno en otro caso. Por tanto después de ejecutar CPI, JP PE saltará sólo cuando BC sea distinto de cero y JP PO lo hará sólo cuando BC=0, con lo que ahorramos tener que mirar explícitamente si BC=0 (si hemos llegado al final del bloque).
Si nos interesara explorar el bloque «hacia atrás», emplearíamos la instrucción «CPD» (comparar con decremento), situando el puntero HL al final del bloque en lugar del principio. La instrucción CPD realiza automáticamente CP (HL), DEC HL, y DEC BC, colocando los flags idénticamente a como lo hace «CPI»,
Pero además tenemos una serie de instrucciones con repetición que constituyen un bucle por sí mismas, simplificando aún más la

cosa. En el ejemplo que estamos analizando, emplearíamos «CPIR» (comparar con incremento y repetición). Es como «CPI» pero vuelve a ejecutarse automáticamente una y otra vez hasta que se dan una de dos condiciones:

—BC=0: Hemos llegado al final del bloque. El flag P/V se pone a cero.
—A=(HL): Hemos encontrado el byte que buscábamos. El flag Z se pone a uno.

La rutina del ejemplo anterior quedará por tanto:

```
INICIO        LD HL,$4000
              LD BC,500 dec.
              LD A,$AA
              CPIR
              JR Z, ENCONTRADO
NO ESTA   . . . . . . .
```

La instrucción «CPIR» realiza las siguientes operaciones:

CP (HL): INC HL:DEC BC hasta que A=(HL) ó BC=0

La instrucción «CPDR» hace lo mismo pero decrementando:

CP (HL): DEC HL:DEC BC hasta que A=(HL) ó BC=0.

Transferencia de bloques

Para trasladar bloques de información de un lugar a otro de RAM se dispone también de instrucciones que facilitan el trabajo. En este caso son necesarios tres elementos:

—Un par de registros como puntero de la dirección del byte del bloque que se va a trasladar.
—Un par de registros como puntero de la dirección de destino del byte trasladado.
—Un registro o par de registros como contador de la longitud del bloque que se está trasladando.

Si tenemos un bloque de 500 bytes a partir de la dirección $4000 y lo queremos trasladar a otro lugar de RAM, por ejemplo a partir de la dirección $5577, podemos construir un bucle, en el que emplearemos el par HL como puntero del byte de origen, DE como puntero del byte de destino y BC como contador de la longitud del bloque:


```
INICIO   LD HL,$4000        inicializamos puntero origen
         LD DE,$5577        ídem. puntero destino
         LD BC,500 dec.     ídem. contador
OTRO     LD A,(HL)          cargamos en A el byte de origen
         LD (DE),A          copiamos A en (DE)
         INC HL             incrementamos puntero origen
         INC DE             incrementamos puntero destino
         DEC BC             decrementamos contador
         LD A,B             probamos si BC ha llegado a
                            cero
         OR C               (ver Cap. 11)
         JR NZ, OTRO        repite para el siguiente byte
HECHO    . . . . . . .
```

Como en el caso de la búsqueda en bloques, podemos utilizar instrucciones que automaticen parte del proceso. «LDI» lleva a cabo por sí mismo las siguientes operaciones:

—Copia el byte cuya dirección es la indicada por el par HL en la dirección indicada por DE, pero sin pasar por A. El contenido de A no se altera al ejecutar «LDI» con lo que ganamos un registro que podremos emplear para otras cosas.
—Decrementa BC. Si BC=0 después de decrementarlo, pone el flag P/V a cero. Si no lo pone a uno.
—Incrementa el par HL.
—Incrementa el par DE.

Según esto la rutina anterior quedará:

```
INICIO   LD HL,$4000
         LD DE,$5577
         LD BC,500 dec.
OTRO     LDI
         JP PE OTRO
HECHO    . . . . . . .
```

La forma de detectar si hemos llegado al final del bloque (si BC=0) es la misma que para la instrucción CPI. Hay que señalar que no existe una instrucción «JR PE e» de salto relativo, lo que obliga a utilizar saltos absolutos. De todas formas disponemos de instrucciones con repetición para trasladar bloques enteros como el caso que nos ocupa: «LDIR» hace el mismo trabajo que «LDI» pero se repite indefinidamente hasta que BC=0 llevando a cabo todo el traslado. Utilizando LDIR la rutina se simplifica al máximo:

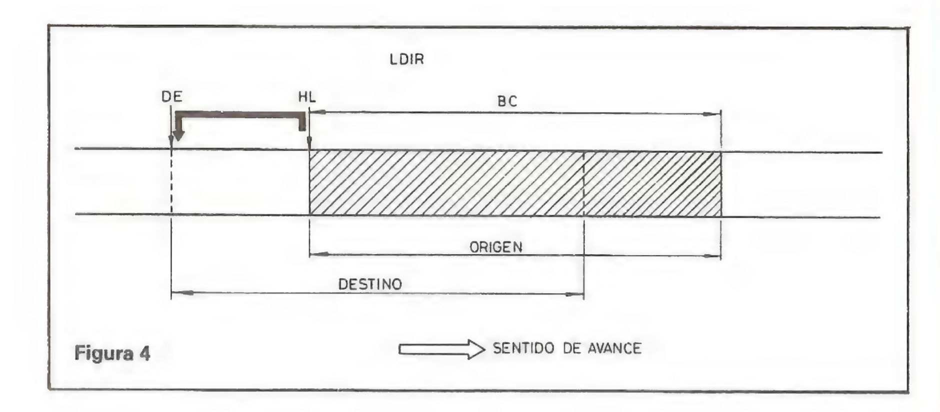


Figura 4

```
INICIO    LD HL,$4000
          LD DE,$5577
          LD BC,500 dec.
          LDIR
HECHO     . . . . . . .
```

Los registros que usan estas instrucciones son fijos, definidos por la instrucción. Esto quiere decir que no podemos emplear HL como contador, sino necesariamente BC, por ejemplo.

La repetición se realiza automáticamente pero el trabajo se lleva a cabo paso a paso; luego podemos desplazar un bloque unos pocos bytes de forma que quede solapado sobre el de origen, como se aprecia en el esquema de la figura 4.

Hay además instrucciones equivalentes pero que decrementan los punteros en lugar de incrementarlos, con lo que el bloque se traslada empezando por el final. He aquí un resumen de todas las instrucciones relacionadas con el traslado de bloques:

MNEMONICO	EFECTO	FLAGS
LDI	(HL) a (DE) INC HL INC DE DEC BC	P/V=0 si BC=0
LDD	(HL) a (DE) DEC HL DEC DE DEC BC	P/V=0 si BC=0

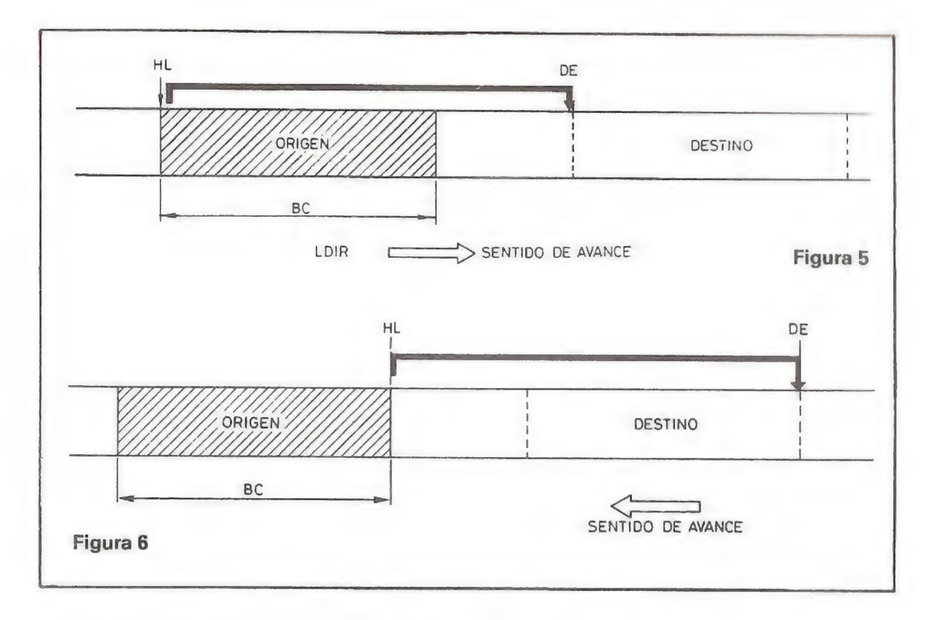


Figura 5

Figura 6

LDIR	igual que LDI pero se repite hasta que BC=0
LDDR	igual que LDD pero se repite hasta que BC=0

CAPÍTULO 13

JUEGO COMPLETO DE INSTRUCCIONES DEL Z80

Este capítulo es un resumen de todas las instrucciones vistas hasta ahora y de algunas que por su poca utilidad en la programación del ZX81 no han sido tratadas, quedando por fin completa la visión de todas las instrucciones de que dispone el Z80. Están ordenadas por orden alfabético y todo el capítulo está concebido para que sirva de consulta rápida ante una duda al realizar un programa.

Para la explicación del funcionamiento se utilizarán los símbolos siguientes:

n: un número de 8 bits (entre $00 y $FF)
nn: un número de 16 bits (entre $0000 y $FFFF)
e: un número de 8 bits (entre $00 y $FF) que se interpretará por la instrucción en complemento a dos, es decir, con signo. Se emplea en instrucciones con saltos relativos. Recuérdese que el salto se cuenta a partir de la siguiente instrucción.

c: condición. Puede ser:

C: si el flag C está a 1
NC: si el flag C está a 0
Z: si el flag Z está a 1 (el resultado fue cero)
NZ: si el flag Z está a 0
PE: si el flag P/V está a 1
PO: si el flag P/V está a 0
M: si el flag S está a 1
P: si el flag S está a 0

x ó y: su significado se especifica en cada caso. El nombre real del mnemónico no es por ejemplo «AND x» sino «AND B». Se emplea la forma con x ó y para poder agrupar de una forma más clara los distintos casos de cada instrucción. El lector encontrará una lista detallada de cada mnemónico con su código en el Apéndice I.

ADC x,y: Cuando «x» es A «y» puede ser (HL), (IX+d), (IY+d), A,B,C,D,E,H,L ó n. Cuando «x» es HL «y» puede ser HL, BC, DE ó SP. Suma «x» más «y» guardando el resultado en «x» y entonces le suma el flag C (carry).

ADD x,y: Cuando «x» es A «y» puede ser (HL), (IX+d), (IY+d), A,B,C,D,E,H,L ó n. Cuando «x» es HL «y» puede ser HL,BC,DE ó SP. Cuando «x» es IX ó IY «y» puede ser BC,DE,SP ó IX si «x» es IX ó IY si «x» es IY. Suma «x» más «y» guardando el resultado en «x». En el ZX81 IX no conviene alterarlo y IY debe contener $4000 al volver el control al intérprete BASIC.

AND x: «x» puede ser (HL), (IX+d), (IY+d), A,B,C,D,E,H,L ó n. Realiza un «Y» lógico, bit a bit, entre A y «x» guardando el resultado en A. El resultado se obtiene en cada bit: 0y0=0, 1y0=0, 1y1=1. Se posicionan los flags de acuerdo con el resultado.

BIT b,x: «b» es el número del bit dentro de «x» que se explora. «x» puede ser A,B,C,D,E,H,L, (HL), (IX+d), (IY+d). El bit «b» de «x» se copia en el flag Z (cero) permitiendo instrucciones condicionales ulteriores.

CALL nn: «nn» es la dirección de memoria a la que salta el programa, tratándose a partir de ahí como una subrutina. La dirección de la siguiente instrucción (la de retorno de la subrutina) se almacena en la pila («stack»). La subrutina debe dejar la pila en el mismo estado al final, para que con una instrucción RET se vuelva al programa principal o al intérprete BASIC.

CALL c,nn: Idéntico a CALL nn pero sólo se ejecuta cuando se cumple la condición «c».

CCF: Complementa el flag C (carry). Si está a uno lo pone a cero y viceversa.

CP x: «x» puede ser (HL), (IX+d), (IY+d), A,B,C,D,E,H,L o n. Compara A con «x» posicionando todos los flags según el resultado. En realidad hace A−«x», sin guardar el resultado en ninguna parte. Afecta a instrucciones condicionales ulteriores.

CPD: Realiza por sí misma «CP (HL)», «DEC HL» y «DEC BC». Si A=(HL) entonces pone el flag Z (cero) a cero. Si BC=0 pone el flag P/V a cero. «JP PE nn» tras CPD saltará sólo si BC es distinto de cero, mientras que «JP PO nn» lo hará sólo si BC=0.

CPDR: Como CPD, pero se repite automáticamente hasta que A=(HL) en cuyo caso el flag Z=0 ó hasta que BC=0 y entonces el flag P/V=0.

CPIR: Como CPDR, pero se incrementa HL en lugar de decrementarlo.

CPI: Como CPD, pero se incrementa HL en lugar de decrementarlo.

CPL: Complementa el registro A de forma que cada uno se convierte en cero y cada cero en uno. No afecta a los flags.

DAA: Ajusta el resultado después de una suma o resta para darlo en decimal, interpretando los dos sumandos en decimal, no en binario. Se usa muy poco.

DEC x: «x» puede ser (HL), (IX+d), (IY+d), A,B,C,D,E,H,L, HL,BC,DE,IX,IY ó SP. Decrementa en uno el valor de «x» y lo almacena de nuevo en «x». Cuando trabaja con registros aislados altera los flags, pero cuando trabaja con parejas de registros no los afecta. El flag C (carry) no se ve afectado en ningún caso.

DI: Inhabilita las interrupciones. A partir del momento en que se encuentra esta instrucción, cuando el Z80 recibe una señal de interrupción la ignora por completo. Esta es una instrucción que no debe utilizarse en el ZX81 porque produce un desastre. En el SPECTRUM impide que se lea el teclado 50 veces cada segundo con lo que se gana algo de rapidez, pero hay que volver a habilitar las interrupciones con EI para leer el teclado o antes de volver al BASIC.

DJNZ e: Decrementa en uno el registro B y si el resultado no es cero salta «e» posiciones hacia adelante o hacia atrás en un salto relativo. Si B=0 continúa con la siguiente instrucción. Es muy útil para construir bucles que deban repetirse un máximo de 256 veces.

EI: Habilita las interrupciones que han sido inhabilitadas previamente con DI. No se usa en el ZX81.

EX (SP),x: «x» puede ser HL,IX ó IY. Se suele emplear con HL para intercambiar los dos bytes de la cima de la pila («stack») con el contenido de HL sin alterar ningún otro registro.

EX x,y: Cuando «x» es AF «y» es AF' (AF prima) y cuando «x» es DE «y» es HL. En el ZX81 el par AF' se emplea por la rutina de DISPLAY y lo mejor es no tocarlo cuando se está en SLOW. EX DE,HL intercambia los contenidos de HL y DE sin alterar ningún otro registro.

EXX: Intercambia HL,BC y DE por HL'BC' y DE' respectivamente, es decir, por los del banco alternativo de registros. Se puede utilizar sin ningún problema en el ZX81, y es muy útil para guardar los registros antes de llamar a una subrutina que los puede alterar.

HALT: Esta instrucción no debe usarse en el ZX81 porque produce un desastre. Mantiene al Z80 realizando NOPs (no operación) indefinidamente hasta que tiene lugar una interrupción.

IM 0, IM 1 e IM 2: Selecciona el modo de interrupción entre los tres posibles en el Z80. Normalmente no se emplean. Unicamente pueden tener utilidad al gestionar periféricos especialmente diseñados, tarea que escapa al objeto de este libro.

IN x,(C): «x» puede ser A,B,C,D,E,H o L. Se utiliza para tomar información del mundo exterior y cargarla en el registro «x». Puede haber hasta 256 aparatos diferentes direccionados cada uno a una vía de acceso («port») que se identifica por un número entre $00 y $FF. Estas vías de acceso («ports») se asignan al diseñar el hardware, por lo que no son utilizables con el equipo standard. Sin embargo el ZX81 tiene asignada la vía de acceso («port») $FE a la entrada MIC y el $FF a la salida EAR y al modulador UHF conjuntamente y se pueden emplear para comunicar con el cassette o para generar sonidos. El dato suministrado por la vía de acceso («port») que se identifica por el contenido del registro C se carga en «x».

IN A,n: Igual que la anterior pero el número de la vía de acceso («port») se suministra directamente como dato.

INC x: «x» puede ser (HL), (IX+d), (IY+d), A,B,C,D,E,H,L, HL,BC,DE,IX,IY y SP. Incrementa «x» en uno almacenando el resultado de nuevo en «x». Cuando «x» es un registro aislado se afecta a todos los flags menos C (carry). Cuando «x» es un par de registros no se afecta a los flags.

IND: Funciona como un IN (HL),(C) y decrementa HL y B, que utiliza como contador para cargar un bloque desde el exterior. No se emplea en el ZX81 standard.

INDR: Funciona como IND pero se repite hasta que B=0. No se emplea.

INI: Funciona como IND pero incrementando HL. No se emplea.

INIR: Funciona como INDR pero incrementando HL. Tampoco se emplea.

JP x: «x» puede ser (HL), (IX), (IY) ó nn. Salta a la dirección de memoria «x» continuando el programa a partir de ahí. Es el equivalente del «GOTO» del BASIC.

JP c,nn: Salta a la dirección de memoria «nn» sólo si se cumple la condición «c» (ver relación al principio del capítulo). Si «c» no se cumple se ignora la instrucción por completo.

JR e: Salto relativo de «e» bytes hacia adelante o hacia atrás interpretando «e» como un número en complemento a dos (ver tabla en Apéndice III) y sumando + 2 al resultado, es decir, contando a partir del inicio de la siguiente instrucción a «JR e». Esto quiere decir que «JR $00» no tiene ningún efecto y que «JR $FE» ($FE= −2 en complemento a dos) entra en un lazo sin fin al saltar indefinidamente sobre sí misma. El margen del salto es de 127 bytes hacia adelante a 128 hacia atrás. Es un byte más corto que «JP nn» y no hay que modificarla si cambiamos la localización del programa. Se recomienda su empleo siempre que sea posible.

JR c,e: «c» sólo puede ser en este caso C,NC,Z ó NZ. Ejecuta un salto relativo como en «JR e» pero sólo en caso de que se cumpla la condición «c».

LD x,y: «x» puede ser A,B,C,D,E,H ó L mientras que «y» puede ser A,B,C,D,E,H,L ó n. Copia el contenido de «y» en «x» destruyendo el contenido previo de «x».

LD A,(xx): «xx» puede ser nn,HL,BC,DE,IX+d ó IY+d. Carga en A el contenido de la dirección de memoria indicada por «xx».

LD x,(y): «x» puede ser A,B,C,D,E,H y L. «y» puede ser HL,IX+d ó IY+d. Carga en «x» el contenido de la dirección de memoria indicada por «y».

LD (xx),y: «xx» puede ser HL,DE,BC,IX+d,IY+d ó nn cuando «y» sea A. Cuando «xx» es IX+d,IY+d ó HL «y» puede ser A,B,C, D,E,H,L ó n. Carga en la dirección de memoria indicada por «xx» el contenido de «y». Cuando «xx» es nn «y» puede ser A,HL,BC, DE,IX,IY ó SP.

LD xx,yy: «xx» puede ser HL,BC,DE,IX,IY ó SP. «yy» puede ser nn,(nn). Carga directamente el dato nn en «xx» ó carga el contenido de las direcciones de memoria nn y nn+1 en «xx» si se utiliza como (nn). Primero se carga el segundo registro (L,C,E...) y luego el primero (H,B,D...) con nn+1.

LD A,x y LD x,A: donde «x» puede ser el registro I especial para interrupciones ó el registro R para refresco de memorias RAM dinámicas. Apenas se utilizan.

LD SP,xx: «xx» puede ser (nn),nn,HL,IX,IY. Carga el puntero de la pila («stack») con el contenido de «xx».

LDD: Carga con decremento. Ejecuta por sí misma y por este orden LD (DE),(HL), DEC HL, DEC DE, y DEC BC. El flag P/V se pone a cero si BC=0 después de decrementarlo. Se emplea para traslado de bloques donde HL se utiliza como puntero en el bloque de origen, DE como puntero de destino y BC como contador de la longitud del bloque a trasladar. El sentido del traslado es desde el final del bloque hacia su inicio. Para repetirla hasta que BC=0 hay que emplear JP PO nn a continuación.

LDDR: Carga con decremento y repetición. Funciona igual que LDD pero se repite automáticamente hasta que BC=0 con lo que se puede trasladar un bloque con una sola instrucción.

LDI: Carga con incremento. Ejecuta por sí misma por este orden LD (DE),(HL), INC HL, INC DE y DEC BC. El flag P/V se pone a cero si BC=0 después de decrementarlo. Se emplea para traslado de bloques donde HL se utiliza como puntero en el bloque de origen, DE como puntero de destino, y BC como contador de la longitud del bloque a trasladar. El sentido del traslado es desde el inicio del bloque hacia su final. Para repetir hasta que BC=0 hay que emplear JP PO nn a continuación.

LDIR: Carga con incremento y repetición. Funciona igual que LDI pero se repite automáticamente hasta que BC=0, con lo que se puede trasladar un bloque con una sola instrucción.

NEG: Trabaja sólo sobre el registro A y tiene como resultado obtener el complemento a dos de A y almacenarlo de nuevo en A. El equivalente en BASIC es «LET A= −A». Por ejemplo, «NEG» seguido de «ADD A,B» es lo mismo que «A=B−A». No afecta a ningún otro registro.

NOP: No operación. Con esta instrucción el Z80 no hace absolutamente nada durante unos tres microsegundos. Se emplea para perder tiempo, generalmente colocándola dentro de un bucle. También es útil para anular instrucciones en un listado sin tener que cambiar las direcciones de memoria de las otras instrucciones. En este caso hay que sustituir cada byte por un NOP.

OR x: «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H,L ó n. Realiza un «O» lógico, bit a bit, entre A y «x», guardando el resultado en A. El resultado se obtiene en cada bit: 0y0=0, 0y1=1, 1y1=1. Se posicionan los flags de acuerdo con el resultado.

OUT (x),y: Cuando «x» es C «y» puede ser A,B,C,D,E,H ó L. Cuando «x» es n «y» sólo puede ser A. Se utiliza para enviar información al exterior contenida en el registro «y» a uno de los 256 posibles aparatos diferentes indicados por el contenido de «x». La asignación de los aparatos a las vías de acceso («ports») se hace al diseñar el hardware, de modo que con el equipo standard sólo se puede emplear la vía de acceso («port») $FE (asignada al conector MIC) y la $FF (asignada al conector EAR y al modulador UHF simultáneamente), empleándose para comunicar con el cassette y para generar sonidos.

OTDR: Funciona como OUT (C),(HL) seguido de DEC HL y DEC B que se emplea como contador, y se repite automáticamente hasta que B=0 después de decrementarlo. No se usa en el ZX81.

OTIR: Como OTDR, pero incrementa HL en lugar de decrementarlo. No se emplea.

OUTD: Como OTDR pero sin repetirse. No se emplea.

OUTI: Como OTIR pero sin repetirse. No se emplea.

POP xx: «xx» puede ser AF,HL,BC,DE,IX,IY. El contenido de la cima de la pila («stack») (los bytes indicados por el puntero SP y por SP+1) se copia en el par «xx» y se aumenta en dos el puntero SP (Recuérdese que la pila o «stack» del Z80 «cuelga cabeza abajo»). El byte indicado por SP se copia en el segundo registro de la pareja y el indicado por SP+2 se copia en el primer registro del par «xx». El contenido previo de «xx» queda destruido al copiar el nuevo. Se utiliza para gestionar la pila («stack») junto a la instrucción PUSH. SP=SP+2.

PUSH xx: «xx» puede ser AF,HL,BC,DE,IX,IY. Se copia el contenido del par «xx» en la cima de la pila («stack»), quedando guardado y permitiendo utilizar el par «xx» para otros propósitos, pudiendo recuperar el contenido anterior con POP «xx». En realidad se copia en (SP−1) el contenido del segundo registro de «xx», en (SP−2) el contenido el primer registro de «xx» y se ejecuta SP=SP−2 (la pila o «stack» del Z80 «cuelga cabeza abajo»). El contenido del par «xx» no se altera.

RES b,x: «b» es el número del bit que se coloca a cero en el byte «x». «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H ó L. El bit indicado por «b» del byte indicado por «x» se pone a cero.

RET: Regresa de una subrutina o devuelve el control al intérprete BASIC. En realidad hace un POP al registro especial PC que es el contador de programa, de modo que si en la cima de la pila («stack») hay otro dato que no sea la dirección de retorno que coloca automáticamente la instrucción CALL, el programa salta a esa dirección de memoria (la indicada por el contenido de la cima de la pila). Esto puede servir para hacer «regresar» una subrutina a un punto diferente al de partida.

RET c: Regresa de la subrutina sólo en el caso de que se cumpla la condición «c». Si no se ignora la instrucción. Por lo demás funciona igual que RET.

RETI: Prevista para finalizar una subrutina que sólo se llama mediante una interrupción. No se utiliza en la programación del ZX81 con el equipo standard.

RETN: Prevista para finalizar una subrutina llamada por una interrupción no enmascarable. Tampoco se emplea con el equipo standard.

RL x: «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H, ó L. Cada bit de «x» se desplaza un lugar a su izquierda ocupando la posición de su vecino. El bit 0 se ocupa con el contenido del flag C (carry) y el bit 7 va a parar al flag C. El resultado afecta a los flags.

RLA: Tiene código $17. No se debe confundir con «RL A» que tiene código «CB07». Aunque ambas tienen el mismo efecto, RLA es más rápida, ocupa un byte menos en la memoria y el resultado no afecta a los flags (salvo C).

RLC x: «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H ó L. Se efectúa una rotación bit a bit de forma que cada bit pasa a ocupar la

posición del de su izquierda, y el bit 7 pasa a ocupar el bit 0 a la vez que se copia en el flag C (carry). El resultado afecta a los flags.

RLCA: Tiene el mismo efecto que «RLC A», pero ocupa un byte menos, es más rápida y el resultado no afecta a los flags (salvo C).

RLD: Los bits 0 a 3 de la dirección de memoria indicada por el contenido del par HL se desplazan a ocupar los bits 4 a 7 de esa misma dirección. El contenido previo de los bits 4 a 7 se desplazan para ocupar los bits 0 a 3 del registro A. Los bits 0 a 3 que contenía A se desplazan para ocupar los bits 0 a 3 de la dirección de memoria especificada al principio. Todos estos cambios ocurren simultáneamente. Para verlo más claro consultar el Cap. 10. Se emplea muy poco. No confundir con «RL D»: no tiene nada que ver.

RR x: «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H ó L. Cada bit de «x» se desplaza un lugar a su derecha ocupando la posición de su vecino. El bit 0 pasa a ocupar el flag (carry) y el contenido previo del flag C se copia en el bit 7. El resultado afecta a los flags.

RRA: Tiene el mismo efecto que «RR A» pero ocupa un byte menos, es más rápida y el resultado sólo afecta al flag C.

RRC x: «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H ó L. Se efectúa una rotación bit a bit de forma que cada bit pasa a ocupar la posición del de su derecha y el bit 0 pasa a ocupar el bit 7, copiándose además en el flag C (carry). El resultado afecta a los flags.

RRCA: Tiene el mismo efecto que «RRC A» pero ocupa un byte menos, es más rápida y sólo afecta el flag C.

RRD: Puesto que los bits 0 a 3 de un byte corresponden al segundo dígito del contenido del byte en Hex. y los bits 4 a 7 corresponden al primer dígito, se puede explicar el efecto de RRD de la siguiente forma: el segundo dígito de (HL) pasa a ocupar el segundo dígito de A. El segundo dígito de A pasa a ocupar el primer dígito de (HL). El primer dígito de (HL) pasa a ocupar el segundo dígito de (HL). Todos estos cambios se realizan simultáneamente. Más detalles en el Cap. 11. No confundir con «RR D»: no tiene nada que ver.

RTS x: Funciona como «CALL» pero salta a una posición de memoria que viene determinada por el código de la instrucción. «x» puede ser $00,$08,$10,$18,$20,$28,$30 ó $38 y es la dirección a la que salta en cada caso. Tiene la ventaja de que ocupa sólo un byte en lugar de los tres necesarios para un «CALL nn», pero no podemos elegir

el lugar del salto. Las direcciones de salto corresponden en el ZX81 a rutinas de ROM y su efecto se explica con detalle en el Cap. 20.

SBC x,y: Cuando «x» es A «y» puede ser (HL),(IX+d),(IY+d), A,B,C,D,E,H,L ó n. Cuando «x» es HL «y» puede ser HL,BC,DE ó SP. Resta «y» de «x» y a continuación resta el contenido del flag C (carry) al resultado, que se guarda en «x». El resultado afecta a los flags.

SCF: Tiene como único efecto poner el flag C (carry) a uno.

SET b,x: «b» es el bit de «x» que se pone a uno. «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H ó L. Pone a uno el bit «b» de «x». El resultado no afecta a los flags.

SLA x: «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H ó L. Cada bit de «x» se desplaza una posición a su izquierda ocupando la de su vecino. El bit 7 se copia en el flag C (carry) y el bit 0 se pone a cero. El resultado afecta a los flags.

SRA x: «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H ó L. Cada bit se desplaza una posición a su derecha ocupando la posición de su vecino. El bit 0 se copia en el flag C (carry) y el bit 7 queda inalterado. El resultado afecta a los flags.

SRL x: «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H ó L. Cada bit se desplaza una posición a su derecha para ocupar la de su vecino. El bit 0 se copia en el flag C (carry) y el bit 7 se pone a cero. El resultado afecta a los flags.

SUB x: «x» puede ser (HL),(IX+d),(IY+d),A,B,C,D,E,H,L ó n. El mnemónico puede ser también «SUB A,x». Resta «x» del registro A guardando el resultado en A. El resultado afecta a los flags. No se dispone de resta de parejas de registros directamente, pero se puede hacer empleando «AND A» seguido de «SBC HL,ss» donde «ss» puede ser HL,BC,DE ó SP.

XOR x:«x» puede ser (HL), (IX+d),(IY+d),A,B,C,D,E,H,L ó n. Ejecuta un «O» lógico exclusivo entre el registro A y «x». El resultado se obtiene en cada bit: 0y0=0, 0y1=1, 1y1=0. «XOR A» tiene el efecto de poner a cero tanto el flag C como el registro A.

CAPÍTULO 14

ESTRUCTURA DE LA PANTALLA: D—FILE

Una vez completada la visión del juego de instrucciones del Z80 y las particularidades que puede haber en su utilización en el ZX81, vamos a ver otra parte fundamental para poder programar en CM: las estructuras de los elementos que lo comunican con el mundo exterior, es decir, con nosotros. En definitiva es a través de estos elementos como podremos ver el resultado de nuestro programa y como podremos influir en él (pulsando teclas ó entrando datos mediante el cassette, por ejemplo).

Empezaremos este análisis con el estudio de la estructura de la pantalla (concretamente de la zona de memoria RAM donde se archiva lo que vemos por la pantalla del TV en un momento dado).

La zona de RAM comprendida entre las direcciones contenidas en las variables de sistema D—FILE y VARS contiene la información que vemos en la pantalla en todo momento. El chip de lógica envía 50 veces cada segundo una interrupción al Z80, que deja lo que está haciendo (guardando todos sus registros en la pila o «stack»). Si el ZX81 está en modo SLOW, se dedica a enviar la información de este archivo de pantalla al modulador UHF en forma de «1» y «0», donde un «1» es un punto negro y un «0» es un punto blanco. Esto se hace con la precisión de tiempo adecuada y con los sincronismos horizontal y vertical necesarios para que en la toma de antena del televisor la información esté codificada exactamente igual que si se tratara de un programa de TVE en el canal 36 de UHF. Junto al chip de lógica participa en esta tarea y en su sincronización en el tiempo, una rutina de ROM que llamaremos DISPLAY, responsable de la inutilización de algunos registros e instrucciones disponibles en el Z80 pero que nos están vedados al programar nosotros el ZX81.

Cuando la pantalla y sus sincronismos está completada, el Z80 recupera todos sus registros de la pila («stack») y sigue con nuestro programa, hasta que tenga lugar una nueva interrupción. Este sistema es uno de los factores que hace posible el bajo coste del ZX81, pero por otra parte es el responsable de su lentitud. En modo FAST el ZX81 es más rápido, sencillamente, porque no se pierde tiempo en enviar información a la pantalla.

Según esto, cuando nosotros o el programa en CM ponemos el código de un carácter en una posición del archivo de pantalla (con una instrucción LD por ejemplo), a partir de ese momento podemos ver el carácter que corresponda a ese código en la pantalla del TV y permanecerá ahí hasta que pongamos otra cosa en esa misma dirección del archivo de pantalla. El problema está en qué byte de los del archivo hay que poner un código para que el carácter aparezca en la posición adecuada en la pantalla. Para solucionar esto debemos conocer la estructura de este archivo.

En realidad puede haber dos estructuras diferentes según tengamos destinado al BASIC (es decir por debajo de la variable de sistema RAMTOP) más de 3 1/4 K de memoria o menos de 3 1/4 K. Para simplificar las cosas hablaremos a partir de ahora de estructura con 1K ó de estructura con 16K, dando por supuesto que la barrera está realmente en 3 1/4K.

Ambas estructuras tienen unas características comunes:

Cada posible posición de la pantalla (32×24=768 posibles posiciones) ocupa un byte en el archivo en principio, estando ordenados de la siguiente forma: primero el de la posición de la esquina superior izquierda, luego por orden de izquierda a derecha todos los de la primera línea (la superior), luego los de la segunda, tercera, cuarta, etcétera, hasta la línea 24 (la inferior) siendo por tanto el último el de la posición de la esquina inferior derecha.

Dicho de otra forma, están colocados en el mismo orden en que aparecen al ejecutar el siguiente programa BASIC:

```
10 POKE 16418,0
20 FOR N= 1 TO 768
30 PRINT «O»;
40 LET L= SIN SIN SIN SIN PI
50 NEXT N
```

El final de cada línea se identifica por un código que utiliza el chip de lógica como marca de fin de línea: el código es 118 dec. ($76).

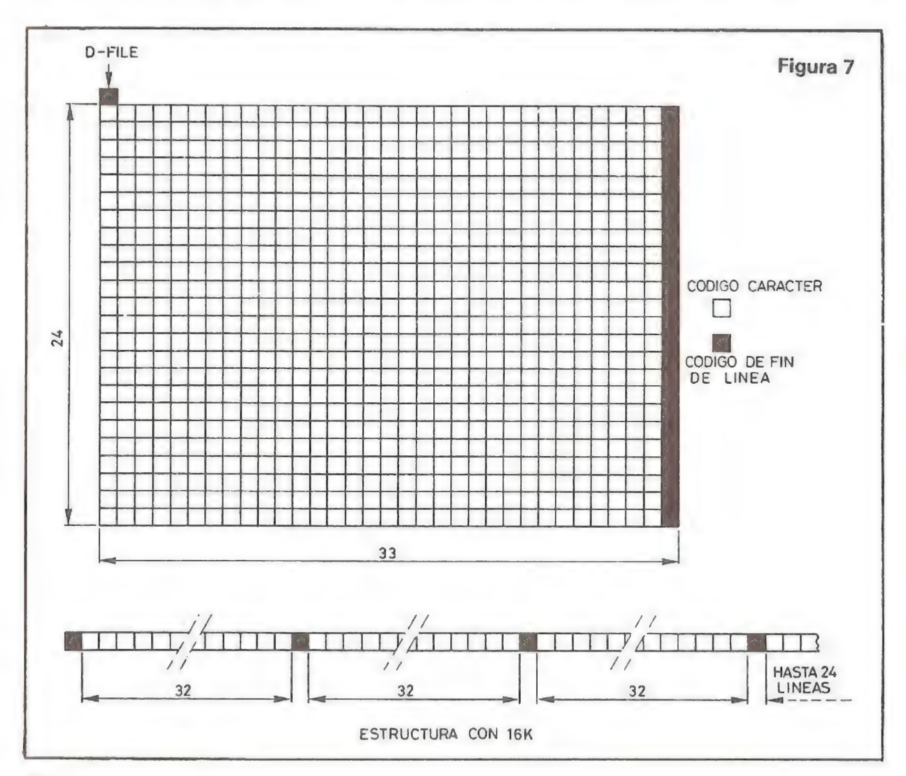
Además el primer byte del archivo debe contener necesariamente una de estas marcas $76 porque, de lo contrario, ocurre un desastre.

Cuando el chip de lógica encuentra como código una marca de fin de línea, organiza las cosas para que el resto de la línea aparezca en blanco y al llegar a la línea siguiente continúa normalmente.

Veamos ahora las diferencias entre la estructura con 1K y con 16K.

14.1. Estructura con 16K (más de 3 1/4K bajo RAMTOP):

La longitud de archivo de pantalla es fija sea cual sea su contenido desde la pantalla vacía hasta cuando está llena por completo. En la dirección indicada por la variable de sistema D—FILE, la primera del archivo, hay un código 118 dec. ($76) imprescindible para que la



cosa funcione. A continuación hay 32 bytes con los códigos de la primera línea, luego una marca de fin de línea ($76), luego 32 bytes para la segunda línea, otra marca de fin de línea y así sucesivamente hasta completar las 24 líneas, tal como puede verse en la figura 7.

Si llamamos D—FILE a la dirección de inicio del archivo, indicada por la variable de sistema D—FILE, F al número de fila y C al número de columna, tal como se numeran en «PRINT AT», la dirección de una posición cualquiera de la pantalla viene dada por la expresión D—FILE + 1 + 33 × F + C.

14.2 Estructura con 1K (menos de 3 1/4K bajo RAMTOP):

Con la estructura que hemos visto el archivo de pantalla ocupa 768 bytes para las posiciones de pantalla, más 24 bytes para las marcas de fin de línea, más 1 byte para el código $76 inicial; es decir, un total de 793 bytes. Cuando no disponemos más que de 1K de memoria RAM en total (1024 bytes), esta estructura apenas nos dejaría espacio para el propio programa y las variables. La solución adoptada es que el archivo de pantalla se compacta de forma que no emplee memoria para los trozos finales de las líneas cuando están en blanco. La longitud del archivo varía según lo que haya en la pantalla. Con la pantalla completamente llena queda la misma estructura que con 16K, pero en cualquier otra ocasión sólo se archiva de cada línea los códigos hasta el carácter de más a la derecha distinto de un espacio, puesto que al encontrar la marca de fin de línea el chip de lógica envía automáticamente espacios hasta completar la línea.

Cuando la pantalla está vacía la estructura del archivo es la siguiente: el código $76 inicial, seguido de 24 marcas de fin de línea. Esta es su configuración mínima.

Cuando el intérprete BASIC tiene que hacer un PRINT en esta estructura, llama primero a una rutina de ROM que mira si el byte donde debe ir el código existe realmente. Si no es así, se crean espacios en lugar requerido, expandiendo la zona mediante un desplazamiento de todo lo que hay por encima en RAM, hasta la zona libre (ver mapa de memoria en Apéndice IV). Una vez tiene la estructura del archivo de pantalla ajustada, coloca el código del carácter en la dirección adecuada, con lo que aparece en pantalla.

En CM este proceso u otro parecido hay que preverlo en el propio programa. Un buen sistema consiste en crear al principio del programa una estructura parecida a la de 16K pero más pequeña, para que nos ocupe menos memoria.

Si necesitamos por ejemplo una «pantalla» de 10 filas por 12 columnas, hay que conseguir la siguiente estructura en el archivo de pantalla: un código $76 inicial, 10 veces; una secuencia de 12 espacios y un código de fin de línea, y 14 códigos de fin de línea seguidos (completando así las 24 líneas).

Con esta estructura, para situar un carácter debajo de otro basta sumar 13 a la dirección del primero (12 bytes por línea más uno del código de fin de línea).

Para expandir la zona del archivo y colocar un carácter podemos aprovechar parte de la rutina de PRINT de ROM, que colocará el código que se encuentre en el registro A en la posición indicada por el par HL, expandiendo previamente un byte toda la parte de RAM entre (HL) y la zona libre. Esto es precisamente lo que necesitamos, devolviendo además el puntero HL incrementado en uno, listo para poner algo en la siguiente posición. La dirección donde empieza este trozo de la rutina de ROM es $083A. Esta rutina destruye el contenido de DE y BC, por lo que habrá que guardarlos en la pila («stack») antes de llamarla y recuperarlos después.

Según esto, para construir una estructura de pantalla como la del ejemplo puede servir la siguiente rutina:

```
INICIO   LD HL,(16396)   En HL el inicio del archivo
         INC HL          Salta la marca inicial
         LD B,10         Contador para filas
LINEA    PUSH BC         guarda contador en pila
                         («stack»)
         LD B,12         contador columnas
CHR      PUSH BC         guarda contador en pila
                         («stack»)
         PUSH DE         guarda DE en pila («stack»)
         LD A,0          espacio (código carácter a
                         pintar)
         CALL $083A      llama a subrutina de ROM
         POP DE          recupera DE de la pila («stack»)
         POP BC          recupera contador columnas
         DJNZ CHR        siguiente carácter
         INC HL          salta marca de fin de línea
         POP BC          recupera contador líneas
         DJNZ LINEA      siguiente línea
         RET             fin
```


La instrucción «SCROLL» del BASIC introduce la línea inferior en forma de una marca de fin de línea solamente, tanto si se dispone de 1K como de más de 3 1/4K. Hay que tener cuidado de no utilizar «SCROLL» si trabajamos con una estructura de pantalla como la del ejemplo.



crash (desastre)

NOTA IMPORTANTE: En teoría se puede poner en el archivo de pantalla cualquier código entre 0 y 255, pero si se pone un código que no corresponda a ningún carácter la rutina de DISPLAY se confundirá y obtendremos como resultado un desastre. Sólo se puede poner códigos entre 0 y 63, ó entre 128 y 191 inclusive, es decir, todos los que tienen el bit 6 a cero. Además la segunda serie corresponde a los caracteres inversos de la primera, de forma que para obtener el inverso de un carácter determinado basta sumarle 128 a su código.

CAPÍTULO 15

EL TECLADO

El principal medio de que disponemos para comunicar información al ZX81 en mitad de un programa es el teclado. Trabajando en BASIC tenemos una serie de limitaciones, por ejemplo que la tecla «SPACE» detiene el programa, o la imposibilidad de interpretar varias teclas simultáneas.

En CM el teclado no se explora a menos que se indique expresamente en el programa. Uno de los responsables de que no podamos sacar una rutina que ha entrado en un lazo infinito más que desconectando la alimentación, es precisamente que no se consulta el teclado.

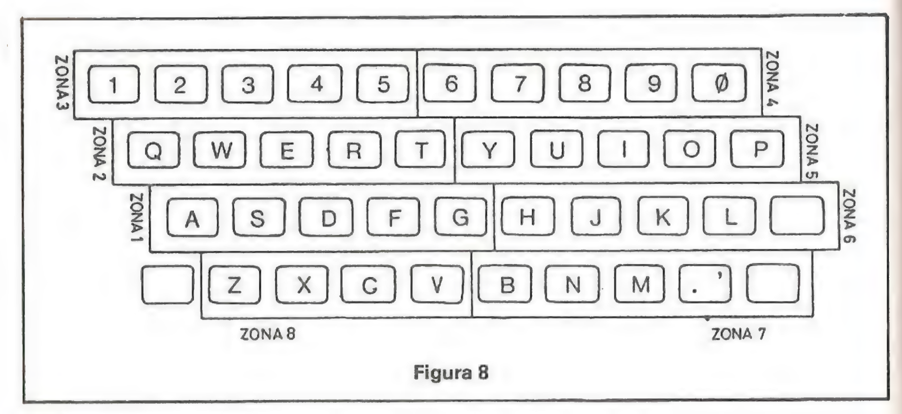


Figura 8

Conociendo como está construido el teclado y la forma de decodificarlo, se puede hacer con él casi cualquier cosa (menos convertirlo en un teclado de verdad). Veamos primero cómo se explora:

Imaginemos el teclado dividido en ocho secciones horizontales que abarcan varias teclas cada una, quedando la tecla SHIFT al margen, tal como muestra la figura 8.

Hay una rutina en la ROM, que llamaremos EXPLORATEC, situada a partir de la dirección de memoria $02BB, que explora cada una de las secciones devolviendo el resultado en el par HL de la siguiente forma:

Cada uno de los ocho bits del registro L representa una de las secciones horizontales en que hemos dividido el teclado. Cuando una tecla de una sección está pulsada en el momento de explorar el teclado, el correspondiente bit del registro L se pone a cero. Si no hay ninguna tecla de las secciones pulsada, todos los bits estarán a uno y L contendrá $FF.

Los valores que puede tener L son los siguientes:

SECCION CON UNA TECLA PULSADA	BINARIO	HEX
Ninguna	11111111	$FF
0	11111110	$FE
1	11111101	$FD
2	11111011	$FB
3	11110111	$F7
4	11101111	$EF
5	11011111	$DF
6	10111111	$BF
7	01111111	$7F

Si hay pulsadas más de una sección a la vez, el registro H tomará el valor correspondiente a poner a cero los bits de las secciones pulsadas.

Con este sistema no se pueden diferenciar teclas dentro de una misma sección. Para solucionar esto se divide el teclado en otras cinco secciones, esta vez verticales (fig. 9).

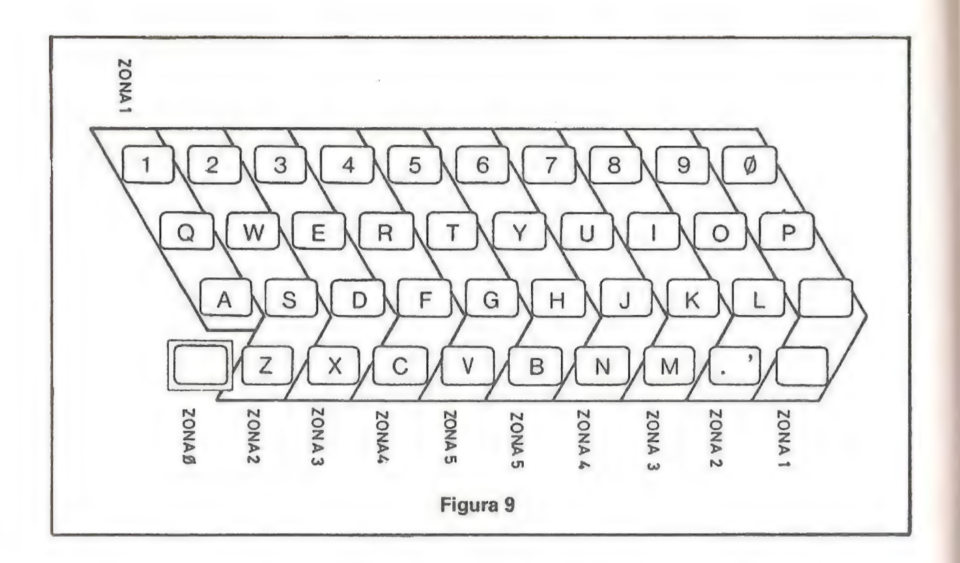


Figura 9

La sección 0 corresponde a la tecla SHIFT aislada. A cada una de las otras secciones corresponden dos columnas de teclas. Al volver de la rutina EXPLORATEC de ROM el registro H puede tomar los siguientes valores, de modo análogo a como se construye el registro L:

SECCION CON UNA TECLA PULSADA	BINARIO	HEX
Ninguna tecla pulsada	11111111	$FF
0 (tecla SHIFT)	11111110	$FE
1	11111101	$FD
2	11111011	$FB
3	11110111	$F7
4	11101111	$EF
5	11011111	$DF

Cuando se pulsa más de una sección el resultado en H se forma combinando los valores anteriores de manera que el bit de cada sección pulsada estará a cero. Los bits 6 y 7 están siempre a uno.

Con este sistema, si pulsamos por ejemplo la tecla «J» y mientras está pulsada se ejecuta la rutina EXPLORATEC, el par HL volverá con el resultado L=10111111 ($BF) y H=11101111 ($EF), es decir, HL=$EFBF. Cualquier tecla dará un resultado diferente, y distinto además si está pulsada junto con SHIFT.

La rutina EXPLORATEC destruye el contenido previo de todos los registros. Cuando se llama a la rutina en medio de otro proceso del que queremos conservar el contenido de los registros, lo que hay que hacer es guardarlos en la pila («stack») antes de llamarla, y recuperarlos después:

```
PUSH HL
PUSH BC
PUSH DE
PUSH AF
CALL $02BB
POP AF
POP BC
POP DE
```

y nos queda HL en la pila («stack») para que lo recuperemos cuando hayamos decodificado el resultado de EXPLORATEC, que está en HL.

Hay otra rutina en ROM que decodifica el extraño valor de HL y lo convierte en la dirección de una tabla de ROM donde están los códigos de los caracteres asignados a las teclas tal como las interpreta el intérprete BASIC; es decir, con o sin SHIFT. Esta rutina, que llamaremos DECODE, está a partir de la dirección de memoria $07BD de ROM, y tiene algunas particularidades:

—El resultado de HL obtenido con EXPLORATEC debe estar en el par BC al entrar en la rutina DECODE.

—Debe haber necesariamente una tecla pulsada al llamar a DECODE, es decir, el resultado de EXPLORATEC debe ser distinto de $FFFF.

Todo esto se puede solucionar aplicando la rutina de esta forma:

```
INICIO    CALL EXPLORATEC   Explora el teclado
          LD B,H            Copia el resultado obtenido
          LD C,L            al par BC
          INC L             ver texto
          LD A,n            n = código que indicará que no
                            hay tecla
          JR Z, NOTECLA     si L = 0 salta a NOTECLA
          CALL $07BD        decodifica dando en HL posición
                            en tabla
          LD A,(HL)         recoge el código de la tabla
                            de ROM
NOTECLA   . . . . . .       sigue el programa
```

Si no hay ninguna tecla pulsada el registro L contendrá $FF, y al hacer INC L entonces L pasará a ser $00 y por tanto el flag Z=1. Como la instrucción «LD A,n» no afecta a los flags, «JR Z,NOTECLA» saltará sólo si L=$00; es decir, si L valía $FF (si no había ninguna tecla pulsada), evitando así entrar en DECODE cuando no se pulse una tecla.

Los códigos que devuelve DECODE corresponden a la tecla pulsada según la lista del apéndice A del Manual de Sinclair, y pueden corresponder a un comando. Por ejemplo si pulsamos SHIFT y «A» simultáneamente, obtendremos en A el código $E3 (227 dec.) que corresponde al comando «STOP».

La ejecución de estas dos rutinas en la forma indicada da el mismo resultado que la sentencia BASIC "LET A=CODE INKEY$"; naturalmente, teniendo en cuenta que aquí A es una variable BASIC, mientras que en CM «A» es el registro A.

15.1. Lectura de varias teclas simultáneas

Con todo lo visto hasta ahora no hemos ganado gran cosa respecto del BASIC, en parte porque utilizamos las mismas rutinas haciendo nosotros de intérprete. Una de las principales ventajas que proporciona el teclado trabajando en CM es la posibilidad de decodificar varias teclas que han sido pulsadas simultáneamente.

Para ello utilizaremos la rutina EXPLORATEC pero no DECODE. Supongamos que queremos decodificar las teclas A,Q,P y L, cada una de las cuales corresponde por ejemplo a la dirección que tomará un objeto determinado, y nos interesa que si se pulsan varias teclas a la vez la dirección que tome sea el resultado de combinar esas direcciones (en diagonal, por ejemplo). No es casualidad que las teclas elegidas correspondan a secciones horizontales del teclado diferentes. Concretamente «A» está en la sección 1, «Q» en la sección 2, «P» en la 5, y «L» en la 6. La solución es muy sencilla. Basta explorar el teclado con EXPLORATEC y luego ir mirando los bits del registro L que correspondan a las secciones de las teclas seleccionadas, llamando en caso de que el bit esté a cero a la rutina que tiene el efecto de la tecla pulsada. Como en las rutinas que producen el efecto correspondiente a las teclas se puede destruir el contenido de L, hay que preservarlo en la pila («stack») antes de que puedan ser llamadas, y recuperarlo después para mirar el siguiente bit:


```
INICIO   CALL $02BB     Llama a EXPLORATEC
         PUSH HL        guarda HL en la pila («stack»)
         BIT 2,L        si es Q, Z=0
         CALL Z,ESQ     si Z=0 llama a la subrutina
                        para Q
         POP HL         recupera HL de la pila («stack»)
         PUSH HL        lo guarda, pero se ha copiado
                        en HL
         BIT 1,L
         CALL Z,ESA
         POP HL
         PUSH HL        hace lo mismo con las otras
         BIT 5,L        teclas
         CALL Z,ESP
         POP HL
         BIT 6,L
         CALL Z,ESL
         RET            fin
```

Con este sistema en realidad se decodifica cualquier tecla de las secciones a las que pertenecen las teclas escogidas, es decir, tanto A como S, D, F ó G darán el mismo resultado, y lo mismo con las de las otras secciones. De todas formas éste es un ejemplo sencillo para ilustrar el sistema. Empleando tanto las secciones horizontales como las verticales y la lógica adecuada se pueden decodificar varias teclas, aunque estén próximas.

Una ayuda para este tipo de decodificaciones consiste en reservar un byte en alguna parte de RAM donde guardar el resultado de EXPLORATEC, y luego ir consultando los bits de este byte (ó bytes) a medida que tengamos que ejecutar los resultados de las teclas. Esto deja libres todos los registros y no obliga a realizar el efecto durante el mismo acto de la exploración, como en el ejemplo.

Capítulo 16

CONEXIONES MIC Y EAR

Nos queda un último elemento de comunicación con el exterior: las conexiones con el cassette. En realidad no tienen por qué conectar con el cassette necesariamente, y se pueden emplear para producir sonido, como veremos más adelante en este mismo capítulo.

El ZX81 se comunica con el cassette por medio de «sonidos», ondas de frecuencia audible (no hay más que oír una cinta grabada con un programa para darse cuenta).

En esencia el sonido no es más que una serie de ondas cuya frecuencia (el número de oscilaciones por segundo) es diferente según que el sonido sea más grave o más agudo. Cuanto mayor es la frecuencia, más agudo es el sonido. Estas oscilaciones se pueden amplificar o transmitir en forma de oscilaciones de corriente eléctrica de las mismas frecuencias que el sonido original. En base a esto es como quedan registradas en un cassette.

Por otra parte los «unos» y «ceros» de un ordenador son paso o ausencia de corriente (concretamente tensión). Cuando en un BUS se lee un número en realidad se «lee» la presencia o ausencia de corriente en las pistas. Una pista con +5 voltios corresponde a un «1» y con 0 voltios corresponde a un «0».

El ZX81 aprovecha estos dos fenómenos para comunicarse con el cassette.

Si vamos enviando por orden, bit a bit, el contenido de RAM a la entrada de un magnetófono, la secuencia de «unos» y «ceros» que no es más que una secuencia de presencia y ausencia de corriente oscilando, es muy parecida a una onda sonora en forma de

corriente eléctrica, como cuando grabamos un disco. Como el cassette no sabe lo que está grabando, registra esa secuencia sin ningún problema, tal como la podemos oír por el altavoz al escuchar la cinta.

Si más tarde ponemos el cassette en PLAY y vamos devolviendo las oscilaciones de corriente a los sucesivos bits de la memoria RAM, en el mismo orden y con idéntica velocidad con que los grabamos, vamos recuperando bit a bit toda la información que habíamos registrado. Esta es en esencia, y salvando todas las cuestiones técnicas, la forma como se comunican el cassette y el ZX81. De hecho, en lugar de manejar impulsos aislados se convierten en dos frecuencias determinadas, pero el sistema aquí descrito funciona perfectamente y es la base de algunos programas para comunicar a velocidades de unos 4000 bits/sg. en el ZX81, en lugar de los 250 bits/sg. standard.

La secuencia de instrucciones:

```
FREC     LD A, frecuencia
PAUSE    DEC A
         JR NZ, PAUSE
         RET
INICIO   LD DE, duración
LAPSO    DEC D
         LD A,D
         OR E
         RET Z
         IN A,($FE)
         CALL FREC
         OUT (FF),A
         CALL FREC
         JR LAPSO
```

produce una nota de frecuencia y duración especificadas en la salida MIC, que se puede conectar a cualquier amplificador o grabar en cinta, o escucharse por el altavoz del televisor, en este caso con algún ruido de fondo. El ZX81 debe estar en FAST para que funcione.

Con esta rutina, un ejercicio interesante puede ser realizar un programa en CM que vaya leyendo y tocando notas de una tabla de frecuencias y duraciones.

CAPÍTULO 17

TECNICAS DE PROGRAMACION

Saber escribir en un idioma no quiere decir ser un buen escritor. Conocer las instrucciones de CM y qué hacen no es suficiente para programar *bien* en CM. Evidentemente es fundamental conocer a fondo el lenguaje que se emplea, pero con las mismas instrucciones se hacen programas buenos y malos. El resultado dependerá en gran parte de la técnica que se emplee al programar.

En primer lugar es muy importante que el programa esté ESTRUCTURADO. Al principio (y a veces no tan al principio) son frecuentes los «programas-spaghetti», donde el curso del proceso va continuamente adelante y atrás, siendo difícil ver sin gran esfuerzo qué hace una parte determinada del listado. Las modificaciones y ajustes en un «programa-spaghetti» son cosa de locos, y al intentar corregir un error se introducen otros.

Una de las reglas de oro para estructurar un programa es evitar los saltos en la medida de lo posible. Al terminar un proceso donde hay saltos es interesante analizar si las instrucciones se pueden disponer en otro orden de forma que se eviten saltos. Esto no quiere decir que en un programa no haya partes comunes que se reutilicen saltando desde otros puntos, pero hay que procurar reducirlas en lo posible, porque de lo contrario el programa pierde claridad.

Con esto está relacionada otra característica que debe tener un programa: debe ser MODULAR. Todo lo relacionado con una misma tarea debe residir junto en el listado en forma de módulos que realizan cada uno una tarea específica y completa. La mejor forma de conseguir modularidad es construir el programa a base de subrutinas

que llevan a cabo cada una de las tareas necesarias, y cuyo orden de ejecución se determina en un lazo principal de instrucciones CALL. Este sistema tiene la ventaja de que en el momento que hay que introducir algo nuevo en el programa sólo hay que escribir la rutina adecuada e introducir un «CALL» a esa rutina en el lugar preciso del lazo principal, sin otras modificaciones. Por otra parte es mucho más fácil detectar los errores en un momento dado, e interpretar una parte del listado sin tener que romperse la cabeza. En la aplicación del Capítulo 19 se pueden ver cómo hacer esto en la práctica.

Otro recurso que da muy buenos resultados es empezar siempre por lo más difícil. Por una parte nos dará una idea de si lo que nos proponemos se puede llevar a la práctica y por otra el resto del programa va quedando condicionado por las características de lo más complicado. Si no, nos podemos encontrar con que a medio confeccionar un programa haya que rehacer lo que ya tenemos, porque no se adapta a algo que sólo se puede hacer de una forma. Al mismo tiempo hay que procurar empezar por aquellos procesos que el programa vaya a utilizar más veces, y que suelen ser además los más complejos; por ejemplo, colocar y desplazar los objetos en la pantalla, las operaciones con los datos, etc. Así aseguramos que gran parte de lo que vamos necesitando ya lo tenemos construido. Se dice que en CM cada bit debe ayudar a colocar el siguiente. Si se emplea el método de las subrutinas se va disponiendo de una serie cada vez mayor de «instrucciones» que facilitan el trabajo y que a su vez formarán parte de nuevas «instrucciones», del tipo de «mover objetos», «exploración del teclado», «calcular la media».

En CM se dispone de muy pocos «registros-variable» donde realizar el proceso. Se hace pues imprescindible disponer de una serie de bytes donde residan las variables, que iremos cargando en registros cada vez que haya que consultarlos o modificarlos. La organización de estas variables internas del programa es importante para que su manejo sea sencillo y eficaz. Hay que procurar poner juntas todas las variables que estén relacionadas, por ejemplo dirección de un objeto en el archivo de pantalla, su velocidad, su código, el sentido de su desplazamiento. Se puede optar por poner juntas todas las variables del programa en una misma zona de RAM o ponerlas antes de cada subrutina. Con la primera técnica hay que dimensionar una zona que sea suficiente antes de empezar, mientras que con la segunda el espacio se va reservando a medida que se necesitan. Una ventaja de la utilización de variables internas es que un parámetro se puede consultar sin dificultad desde cualquier punto del programa. Además de las variables con los parámetros propios del problema que se está tratando,

habrá que disponer de variables que hagan referencia a estados internos del programa en un momento dado, por ejemplo un indicador de si el resultado de una operación que se ha realizado con determinados datos hay que sumarlo a un total parcial o no. Este tipo de variables de uso interno hace posible que las subrutinas sean más versátiles, al poder hacer unas cosas u otras en función del estado de determinada variable.

Al empezar el planteamiento de un programa, conviene dividirlo en partes que se puedan abordar de forma independiente, o con la menor dependencia posible, y luego subdividirlas en otras más concretas, y así sucesivamente hasta que se llegue a un nivel en que ya se puede empezar a programar. Este orden es el inverso al que se sigue para construir las subrutinas. Con esto conseguimos mantener una visión global del programa en todo momento y evitamos el riesgo de obcecarnos en una parte de una subrutina perdiendo de vista el proceso en general.

Aunque desde el punto de vista de la estructura suele ser fatal, cuando el problema de la memoria disponible se hace crítico y necesitamos dos subrutinas muy parecidas que sólo difieren en unas pocas instrucciones, una solución a adoptar es escribir una de las rutinas y otra rutina que lo que haga sea modificar las instrucciones que deben ser diferentes en la rutina listada, colocando en las direcciones adecuadas los códigos de las instrucciones nuevas, llame a la subrutina así modificada y luego restituya la forma original volviendo a colocar los códigos originales en su lugar. Esta solución es sólo para casos de emergencia, pero puede sacar de más de un apuro.

17.1. Criterios para usar los registros adecuados

Parte de la eficacia de una rutina depende de que los registros que se emplean para cada cosa sean los adecuados. Es evidente que una rutina que emplee como pares de registros BL y HC será mucho menos eficaz que si utiliza BC y HL, sencillamente porque el juego de instrucciones del Z80 está orientado para dar facilidades a las parejas BC y HL en lugar de BL y HC.

De modo análogo, unos registros son más versátiles que otros. Interesa siempre cargar en los registros más versátiles los datos con los que haya que hacer mayor número de cosas diferentes. Si comparamos los registros en función de las instrucciones que pueden utilizarlos, veremos que el registro A es el más favorecido. Este registro recibe incluso el nombre especial de «acumulador». Los microprocesadores más sencillos sólo tienen uno o dos acumuladores como registros para manejar datos. A medida que han ido apareciendo nuevos modelos se ha ido ampliando el banco de registros, pero el rey sigue siendo el acumulador.

En cuanto a parejas de registros, el par HL se destaca también como el que tiene más posibilidades. Esto hace que sea ideal para emplearlo como puntero, porque es el único que puede proporcionar la dirección de memoria cuyo contenido se puede cargar en cualquier registro. «LD B,(HL)» existe, mientras que «LD B,(DE)», no.

El registro B utilizado aisladamente y el par BC, son los más versátiles para utilizarlos como contadores junto a instrucciones automáticas o semiautomáticas como «DJNZ e» ó «LDDR». En muchos casos su empleo es obligatorio.

Cuando un registro o una pareja son necesarios para otra tarea con datos diferentes a los que contienen, lo mejor es guardar su contenido en la pila («stack») y recuperarlo luego; ó guardar momentáneamente su contenido en otros registros que no se utilicen en ese momento. De todas formas, el uso racional de los registros hace que estos intercambios se reduzcan al mínimo, ganando en rapidez y ahorro de memoria. En este sentido es muy útil la instrucción «EX DE,HL» que intercambia los contenidos de estas dos parejas dando opción a los datos contenidos originariamente en DE a todas las posibilidades que ofrece el par HL.

La pila ó «stack» es una herramienta muy potente si se combina con una utilización adecuada de los registros y se emplea racionalmente. No obstante no conviene abusar de ella para almacenar datos porque entonces su principal característica, el hecho de que sólo es accesible el último dato almacenado, se convierte en una dificultad y suele ser peor el remedio que la enfermedad.

CAPÍTULO 18

APLICACION PRACTICA: JUEGO «COMECOCOS»

Para completar todo lo visto hasta ahora y al mismo tiempo ver sobre el terreno cómo se convierte una idea de programa en una lista de códigos que hace precisamente lo que queríamos que hiciera, vamos a ver en profundidad un ejemplo práctico de cómo se realiza un programa en código máquina. No se trata de dar sencillamente una lista de instrucciones como si fuera una receta mágica y espectacular, sino de ver con detalle todos los pasos seguidos desde el planteamiento de la idea hasta la colocación del último código.

El programa escogido es un juego: el ya clásico COMECOCOS, donde una especie de cosa con boca va tragando puntos en un laberinto, perseguido por unos fantasmas a los que debe evitar. Está realizado íntegramente en CM, lo que da ocasión de ver cómo se montan todas las partes de exploración de teclado, impresión en pantalla, etc. Cabe en 1K de RAM aprovechando la memoria disponible al máximo. Esto tiene una serie de ventajas: no se necesita ninguna ampliación de memoria, el listado es corto y se puede explicar en profundidad. Además, al programar con la memoria limitada obliga a aprovechar al máximo el espacio y las instrucciones y, en definitiva, a programar bien. Es un ejercicio que recomiendo a quien quiera hacer ejercicios de perfeccionamiento.

Para poder introducirlo en 1K de RAM (incluidas las variables del sistema del ZX81, las variables del propio programa, el archivo de pantalla, las pilas ó «stacks» y el mismo listado) he tenido que

suprimir toda la parte de los contadores de puntos y de persecución de los fantasmas por el «hombre» porque literalmente NO CABEN. Aún así se podrá ver que el programa es muy completo.

bicho (maldito error)



Una advertencia previa: no hay errores en el listado, todos los códigos están bien. Como resultado de la experiencia tenida con libros ingleses (de algo tiene que servir llevar un cierto retraso en estos temas), los listados son copia fotomecánica de los originales. Una vez realizado el programa y listados los códigos, se ha vuelto a entrar a partir del mismo listado que aparece en estas páginas y ha funcionado correctamente. Por tanto, si después de entrar los códigos el programa no funciona, existe la seguridad absoluta de que el problema no está en el listado. Es muy fácil tener un uno por mil de porcentaje de error al entrar los datos con el teclado del XZ81, y con un sólo error basta para que el resultado sea nulo. Si hay algún problema, lo mejor es repasar los códigos entrados y compararlos con el listado. Un error es la única explicación. He puesto el máximo interés en dar seguridad en los listados porque si se pierde la confianza en un libro se hace muy difícil aprender.

18.1. Planteamiento del problema

Lo primero que hay que hacer es acotar el problema y plantearlo globalmente pero con precisión. Conviene hacer siempre el programa lo más completo posible en su planteamiento, aunque algunas cosas no estén muy claras en su forma de llevarlas a cabo. Siempre hay tiempo de simplificar y reducir.

Lo que nos proponemos es un juego con un laberinto cerrado por todas partes menos por dos puntos extremos que comunican entre sí de forma que todo lo que salga por la derecha entra automáticamente por la izquierda y viceversa.

Este laberinto está lleno de puntos que irán desapareciendo a medida que el «hombre» los vaya comiendo, pero que no se verán afectados por el paso de los fantasmas.

En el centro del laberinto hay una zona donde aparecerán los fantasmas al principio del juego y cada vez que el «hombre» sea atrapado.

Hay tres fantasmas que se desplazan al azar dentro del laberinto, escogiendo una dirección y siguiendo recto hasta encontrar un obstáculo. Deben detectar cuándo atrapan al «hombre» y no deben superponerse.

Hay un «hombre» que se desplaza por el laberinto controlado por cuatro teclas de dirección y que va comiendo comida. Debe evitar a los fantasmas y comer todos los puntos del laberinto, momento en que aparecerá de nuevo el laberinto lleno otra vez con puntos. En cada partida se dispone de tres «hombres» en total. Después del último, el juego termina.

Con esto tenemos definido el problema. Ahora necesitamos dividirlo en una serie de bloques independientes que lleven a cabo las distintas tareas. Podemos definir los siguientes bloques:

—Un bloque que pone el laberinto en pantalla, copiando su estructura de alguna parte.

—Otro que pone los fantasmas y el «hombre» en sus posiciones de salida.

—Otro que mueve los fantasmas al azar dentro del laberinto, de acuerdo con las normas del planteamiento.

—Otro que mueva el «hombre» en una dirección hasta que desde el teclado se indique un cambio de dirección, o hasta que se encuentre con un obstáculo.

—Otro que explore el teclado para determinar qué dirección debe tomar el «hombre» en un momento dado.

—Otro que controle los finales; es decir, que actúe en consecuencia cuando el «hombre» es atrapado, cuando se ha completado un laberinto y cuando después de tres «hombres» el juego termina.

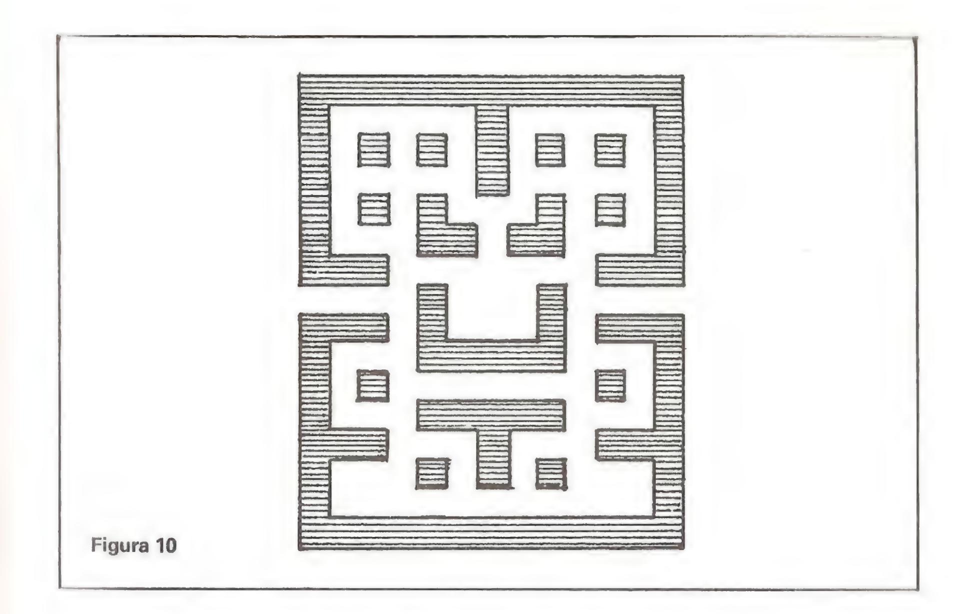
Después de esta etapa ya tenemos un poco más claro lo que necesitamos, aunque no demasiado por el momento. Lo que sí podemos hacer es perfilar las rutinas que necesitaremos:

—PONE LABERINTO

—PONE FANTASMAS Y HOMBRE

—MUEVE FANTASMAS regida por otra de CONTROL DE MTO DE FANTASMAS

—RAND: generador de números aleatorios para los movimientos de los fantasmas, que dependerá de MUEVE FANTASMAS


Figura 10

—MOVIMIENTO TIPO: Movimiento del «hombre» de acuerdo con su situación en el laberinto y las teclas pulsadas.

—TECLADO: Exploración y decodificación del teclado para determinar la posible dirección que deberá tomar el «hombre», si es posible.

—CONTROL DE FINALES

—LAZO PRINCIPAL DE DESARROLLO: Irá llamando a las subrutinas y establecerá la secuencia de la acción.

Ahora ya podemos empezar con la verdadera realización del programa, rutina por rutina, pero sin perder de vista el problema global.

18.2. Diseño del laberinto

Tenemos la memoria limitada a 1K para todo y por tanto el laberinto no podrá ser muy grande, porque una pantalla ya ocupa en el archivo más de 700 bytes. Hay que adoptar una solución de compromiso entre el tamaño y la memoria ocupada al aparecer en pantalla. En consecuencia he escogido un laberinto de 13 por 16 caracteres, con el diseño de la figura 10.

Cuando esté en pantalla ocupará 208 posiciones de memoria, y no nos podemos permitir desperdiciar otras 208 para guardar el patrón del dibujo.

Como el laberinto está formado únicamente por cuadros blancos y negros (en realidad el de CHR$ 8 y el del punto invertido) el dibujo se puede guardar en forma de bits, donde un uno corresponda a cuadro negro y un cero a cuadro blanco. Además, como el laberinto es simétrico, sólo habrá que guardar la mitad de cada línea y luego pintarla de izquierda a derecha y de derecha a izquierda saltando el carácter (bit) central, que no se repite.

Según esto los datos de todo el laberinto quedan reducidos a:

TABLA DATOS LABERINTO

HEX	BINARIO	DEC
FE	11111110	254
82	10000010	130
AA	10101010	170
82	10000010	130
A8	10101000	168
8C	10001100	140
EØ	11100000	224
Ø8	00001000	8
E8	11101000	232
8E	10001110	142
AØ	10100000	160
8E	10001110	142
E2	11100010	226
8A	10001010	138
8Ø	10000000	128
FE	11111110	254

Mirando los unos se puede ver el patrón de la mitad izquierda del laberinto. Así el patrón sólo nos ocupa 16 bytes en lugar de 208. Colocaremos estos datos en una tabla al principio de la primera línea REM, a partir de la dirección 16514.

Ahora necesitamos una rutina que vaya leyendo cada bit de la tabla y pintando uno u otro carácter según sea uno o cero. Para poner el código en el archivo de pantalla utilizaremos una parte de la rutina de PRINT de ROM que expande el archivo un byte y coloca el código,

en la posición indicada por el par HL y con el código en A, dejando HL incrementado en uno. La dirección de esta rutina es $083A, y hay que guardar el par DE en la pila («stack») porque destruye su contenido. Corresponde a la etiqueta CONT del listado.

Utilizaremos el par HL como puntero en el archivo de pantalla, el par DE como puntero en la tabla, B como contador de líneas y rotaciones del byte, y C para almacenar y rotar el byte leído de la tabla mientras se va decodificando. El lector debe dibujar en papel cuadriculado un esquema de la tabla y del archivo de pantalla donde cada cuadro represente un byte, y seguir paso a paso cada instrucción del listado para asegurar que comprende su funcionamiento.

PONE LABERINTO

```
2A ØC 4Ø    PONELAB   LD HL,(16396)
23                    INC HL
Ø6 1Ø                 LD B, 16
11 82 4Ø              LD DE, 16514
C5          LINEA     PUSH BC
1A                    LD A,(DE)
4F                    LD C,A
Ø6 Ø7                 LD B,7
CB Ø1       OTRO      RLC C
C5                    PUSH BC
CD BC 4Ø              CALL PINTA
C1                    POP BC
1Ø F7                 DJNZ OTRO
Ø6 Ø6                 LD B,6
CB Ø9                 RRC C
CB Ø9       OTRO2     RRC C
C5                    PUSH BC
CD BC 4Ø              CALL PINTA
C1                    POP BC
1Ø F7                 DJNZ OTRO2
23                    INC HL
C1                    POP BC
13                    INC DE
```

```
1Ø EØ                DJNZ LINEA
C9                   RET
38 Ø4      PINTA     JR C BLOQUE
3E 9B                LD A,27
18 Ø2                JR CONT
3E Ø8      BLOQUE    LD A,8
D5         CONT      PUSH DE
CD 3A Ø8             CALL $Ø83A
D1                   POP DE
C9                   RET
```

La rutina empieza en la dirección $4092 de RAM.

Una vez solucionado el problema del laberinto vamos a ver cómo poner los tres fantasmas y el hombre en sus posiciones de partida y con la dirección inicial adecuada. Para cada fantasma necesitamos tres variables internas del programa que nos indiquen:

—posición en el laberinto: dirección en el archivo de pantalla donde se encuentra. Dos bytes.

—Tampón: El fantasma tiene que pasar por encima de la comida sin comérsela. Aquí guardaremos el código del carácter que había antes de que el fantasma ocupara esa posición, para restablecerlo cuando se haya ido. Un byte.

—Dirección: Un número nos indicará en qué dirección se está desplazando. Adoptamos el siguiente sistema:

0: Hacia arriba
1: Hacia abajo
2: Hacia la derecha
3: Hacia la izquierda.

El hombre no necesita la variable «Tampón» porque sí va comiendo los puntos, y cuando se vaya de un carácter sencillamente pondremos un cuadro negro.

Estas variables las colocaremos en una zona que tiene libre el ZX81 en sus variables de sistema propias. Se trata de PRBUFF, que va desde las direcciones 16444 a 16476 inclusive. Se utiliza como tampón para pasar una línea a la impresora y cuando no se emplea están vacías. Se borran al pulsar NEWLINE, pero durante el programa esto no ocurrirá, así que tenemos 32 bytes a nuestra disposición (el 33 es NEWLINE). Las direcciones para estas variables internas de nuestro programa serán:

16444—16445:	Posición fantasma 1
16446:	Dirección fantasma 1
16447:	Tampón fantasma 1
16448—16449:	Posición fantasma 2
16450:	Dirección fantasma 2
16451:	Tampón fantasma 2
16452—16453:	Posición fantasma 3
16454:	Dirección fantasma 3
16455:	Tampón fantasma 3
16456—16457:	Posición tipo
16458:	Dirección tipo

La posición inicial de los tres fantasmas está a 118 bytes del inicio del archivo de pantalla y 119 y 120 respectivamente, y la del tipo a 147 bytes. Esto se calcula contando los bytes en un papel con el dibujo del laberinto, teniendo en cuenta los códigos de fin de línea. Para poner en la pantalla a los fantasmas y al tipo la primera vez tenemos que inicializar las variables que hemos visto, los fantasmas con dirección 0 (hacia arriba) y el tipo con dirección 3. Emplearemos el par HL como puntero en la zona de variables y en el par DE pondremos las direcciones iniciales una vez calculadas a partir del inicio del archivo de pantalla:

PONE FANTASMAS

```
2A ØC 4Ø   PONE      LD HL,(16396)
11 76 ØØ             LD DE,118
19                   ADD HL,DE
11 3C 4Ø             LD DE,16444
EB                   EX DE,HL
Ø6 Ø3                LD B,3
C5                   PUSH BC
73         UNO       LD (HL),E
23                   INC HL
72                   LD (HL),D
23                   INC HL
```

```
36 ØØ          LD (HL),Ø
23             INC HL
3E 8Ø          LD A,128
77             LD (HL),A
23             INC HL
13             INC DE
C1             POP BC
1Ø FØ          DJNZ UNO
```

PONE HOMBRE

```
2A ØC 4Ø       LD HL,(16396)
11 93 ØØ       LD DE,147
19             ADD HL,DE
22 48 4Ø       LD (16456),HL
3E Ø3          LD A,3
32 4A 4Ø       LD (16458),A
C9             RET
```

Los fantasmas se mueven en la misma dirección hasta que encuentran un obstáculo (código 8) o al tipo (código 180). Si salen por un pasillo deben entrar por el otro. Si han tropezado con una pared deben cambiar su dirección al azar, determinada por la subrutina RAND que veremos más adelante. La rutina MUEVE FANTASMAS calcula la posición que debe ocupar el fantasma si hiciera el movimiento según la dirección indicada por la variable interna. Mira qué hay en esa nueva posición y según lo que encuentre obra en consecuencia, saltando a la parte adecuada de la rutina:

Si hay pared recupera de la pila («stack») la posición de antes de intentar el movimiento, coloca mediante la rutina RAND otra dirección al azar y vuelve al principio de MUEVE para intentarlo de nuevo, hasta encontrar por tanteo una dirección de desplazamiento válida.

Si no hay pared mira si está el tipo. Si está ahí coloca al fantasma en su lugar y en la posición dejada por el fantasma coloca el carácter del tampón, saltando a la rutina MUERTO.

Si no está el tipo mira si va a salir por un pasillo lateral (código 118 de fin de línea). Si va a salir calcula la nueva posición en el otro lado del laberinto sumando o restando 12, según que la dirección sea hacia la izquierda o hacia la derecha, y salta a NOPARED para ver si en la nueva posición está el tipo y sigue a partir de ahí.

Si no salta mira si la posición que va a ocupar está ya ocupada por otro fantasma. Si es así cambia su dirección al azar con RAND y se da por terminado el movimiento esta vez.

Si no hay otro fantasma es que la nueva posición está libre, porque ya ha comprobado que no hay ningún otro objeto posible. Entonces pone los nuevos datos en las variables internas de ese fantasma, restableciendo el carácter tampón en su posición de pantalla y coloca el fantasma en la nueva posición de la pantalla, dando el movimiento por terminado.

MUEVE FANTASMAS

```
E5           MUEVE     PUSH HL
5E                     LD E,(HL)
23                     INC HL
56                     LD D,(HL)
23                     INC HL
7E                     LD A,(HL)
D5                     PUSH DE
EB                     EX DE,HL
11 ØE ØØ               LD DE,14
FE ØØ                  CP Ø
C2 6A 41               JP NZ NOSUBE
A7                     AND A
ED 52                  SBC HL,DE
7E           COMUN2    LD A,(HL)
FE Ø8                  CP 8
2Ø ØB                  JR NZ NOPARED
E1                     POP HL
E1                     POP HL
23                     INC HL
23                     INC HL
CD 94 41               CALL RAND
2B                     DEC HL
2B                     DEC HL
```

```
18 DD                    JR MUEVE
FE B4       NOPARED      CP 18Ø
2Ø 1Ø                    JR NZ NOTIPO
36 ØB                    LD (HL),11
C1                       POP BC
EB                       EX DE,HL
E1                       POP HL
73                       LD (HL),E
23                       INC HL
72                       LD (HL),D
23                       INC HL
23                       INC HL
7E                       LD A,(HL)
Ø2                       LD (BC),A
F1                       POP AF
C3 69 42                 JP MUERTO
FE 76       NOTIPO       CP 118
2Ø 17                    JR NZ NOSALTA
E1                       POP HL
D1                       POP DE
D5                       PUSH DE
E5                       PUSH HL
13                       INC DE
13                       INC DE
1A                       LD A,(DE)
11 ØC ØØ                 LD DE,12
FE Ø2                    CP 2
2Ø Ø5                    JR NZ ENTRADCHA
A7                       AND A
ED 52                    SBC HL,DE
18 Ø1                    JR COMUN
19          ENTRADCHA    ADD HL,DE
7E          COMUN        LD A,(HL)
18 D1                    JR NOPARED
```



```
FE ØB       NOSALTA      CP 11
2Ø Ø8                    JR NZ NADA
D1                       POP DE
E1                       POP HL
23                       INC HL
23                       INC HL
CD 94 41                 CALL RAND
C9                       RET
E3          NADA         EX (SP) HL
C1                       POP BC
D1                       POP DE
D5                       PUSH DE
13                       INC DE
13                       INC DE
13                       INC DE
1A                       LD A,(DE)
F5                       PUSH AF
ØA                       LD A,(BC)
12                       LD (DE),A
F1                       POP AF
77                       LD (HL),A
E1                       POP HL
71                       LD (HL),C
23                       INC HL
7Ø                       LD (HL),B
3E ØB                    LD A,11
Ø2                       LD (BC),A
C9                       RET
FE Ø1       NOSUBE       CP 1
2Ø Ø3                    JR NZ NOBAJA
19                       ADD HL,DE
18 99                    JR COMUN2
FE Ø2       NOBAJA       CP 2
2Ø Ø3                    JR NZ NODCHA
```

```
23                   INC HL
18 92                JR COMUN2
2B        NODCHA     DEC HL
18 8F                JR COMUN2
```

Como esto debe hacerse para cada uno de los tres fantasmas, implementamos otra rutina CONTROL MTO FANTASMAS que va colocando en HL la dirección donde empiezan las variables internas para cada fantasma y llama a MUEVE. Luego llama a RAND una vez sin colocar el resultado en ningún sitio (en la dirección 0 de ROM, lo que es imposible) para hacer avanzar una posición la secuencia de números «aleatorios» y hacer que sea un poco más aleatoria.

CONTROL MTO FANTASMAS

```
21 3C 4Ø  MTOFAN     LD HL,16444
CD F7 4Ø             CALL MUEVE
21 4Ø 4Ø             LD HL,16448
CD F7 4Ø             CALL MUEVE
21 44 4Ø             LD HL,16452
CD F7 4Ø             CALL MUEVE
21 ØØ ØØ             LD HL,Ø
CD 94 41             CALL RAND
C9                   RET
```

La rutina RAND, de la que ya hemos hablado, toma un número de 16 bits de la variable SEED del sistema del ZX81 (dirección 16434—16435); lo multiplica por 41; deja otra vez el nuevo resultado en SEED y toma, como número, los tres primeros bits del byte más significativo (bits 8, 9 y 10 del número de SEED) como un número al azar, entre 0 y 3. Después, coloca este número en la dirección de memoria indicada por el par HL al entrar en la rutina (se guarda en la pila o «stack»). Este sistema da una serie de números que se pueden considerar aleatorios para nuestras necesidades.

RAND (HL puntero en dir.)

```
E5        RAND       PUSH HL
2A 32 4Ø             LD HL,(16434)
```



```
54                   LD D,H
5D                   LD E,L
29                   ADD HL,HL
29                   ADD HL,HL
19                   ADD HL,DE
29                   ADD HL,HL
29                   ADD HL,HL
29                   ADD HL,HL
19                   ADD HL,DE
22 32 4Ø             LD (16434),HL
7C                   LD A,H
E6 Ø3                AND 3
E1                   POP HL
77                   LD (HL),A
C9                   RET
```

El tipo se mueve con un sistema muy similar al de los fantasmas, pero los cambios de dirección vendrán determinados por la rutina TECLADO en función de las teclas que se pulsen, cuya dirección colocará en la variable «Dirección tipo». La rutina MOVIMIENTO TIPO se encargará sólo de desplazarlo en la dirección que contenga esa variable cuando no choque con una pared. Mirará también si sale por el pasillo lateral y si es comido por un fantasma.

Primero se calcula la nueva posición en función de la dirección cuyo código está en 16458 y una vez calculada se pasa a una parte común (COMUN 3) donde mira si en esa nueva posición hay pared (código 8). Si es así, la rutina termina sin hacer nada más; si no, mira si hay un fantasma (código 11) en cuyo caso salta a la rutina MUERTO; si no, mira si hay comida (código 155). Si hay salta a COME, donde se incrementa un contador de puntos situado en la dirección 16459. Si ha comido 105 puntos habrá vaciado el laberinto, y entonces salta a la rutina FINCUADRO, que restablecerá la comida en el laberinto. Si no se ha llegado a 105 puntos comidos salta a la parte final de la rutina: NOSALTA2.

En el caso de que no se haya encontrado comida mira si va a salir por un pasillo lateral. Si va a salir se calcula su nueva posición al otro extremo del laberinto como cuando saltan los fantasmas, sumando o restando 12 según la dirección de movimiento.

Si no va a salir por el pasillo es que está pasando por un sitio

sin comida y no va a tropezar con nada porque ya se han comprobado todas las posibilidades. Entonces en la posición anterior (guardada en BC) se pone un cuadro negro (código 128) y en la nueva el tipo (código 180) y se actualiza su variable de posición, terminando la rutina.

MOVIMIENTO TIPO

```
2A 48 4Ø  MTOTIPO     LD HL,(16456)
3A 4A 4Ø              LD A,(16458)
44                    LD B,H
4D                    LD C,L
11 ØE ØØ              LD DE,14
FE ØØ                 CP Ø
2Ø 3C                 JR NZ NOSUBE2
ED 52                 SBC HL,DE
7E        COMUN3      LD A,(HL)
FE Ø8                 CP 8
C8                    RET Z
FE ØB     COMPRUEBA   CP 11
CA 69 42              JP Z MUERTO
FE 9B                 CP 155
28 2Ø                 JR Z COME
FE 76                 CP 118
2Ø 13                 JR NZ NOSALTA2
3A 4A 4Ø              LD A,(16458)
11 ØC ØØ              LD DE,12
FE Ø2                 CP 2
2Ø Ø5                 JR NZ ENTRADCHA2
A7                    AND A
ED 52                 SBC HL,DE
18 Ø1                 JR COMUN4
19        ENTRADCHA2  ADD HL,DE
7E        COMUN4      LD A,(HL)
18 EØ                 JR COMPRUEBA
3E 8Ø     NOSALTA2    LD A,128
Ø2                    LD (BC),A
```



```
36 B4                 LD (HL),18Ø
22 48 4Ø              LD (16456),HL
C9                    RET
3A 4B 4Ø  COME        LD A,(16459)
3C                    INC A
FE 69                 CP 1Ø5
28 65                 JR Z FINCUADRO
32 4B 4Ø              LD (16459),A
18 EA                 JR NOSALTA2
FE Ø1     NOSUBE2     CP 1
2Ø Ø3                 JR NZ NOBAJA2
19                    ADD HL,DE
18 BF                 JR COMUN3
FE Ø2     NOBAJA2     CP 2
2Ø Ø3                 JR NZ NODCHA2
23                    INC HL
18 B8                 JR COMUN3
2B        NODCHA2     DEC HL
18 B5                 JR COMUN3
```

Nos falta leer el teclado y colocar el código de dirección en 16458. Para ello empleamos el sistema descrito en el capítulo sobre el teclado, en la rutina TECLADO: exploramos el teclado con la rutina de ROM EXPLORATEC de $02BB y adaptamos el resultado para decodificarlo con la rutina DECODE en $07BD. Entonces lo comparamos con los códigos que corresponden a los números pintados en las teclas de dirección y cuando coinciden ponemos el código en dirección adecuada en la variable «dirección tipo». Entre la exploración del teclado y la decodificación hay una llamada a RAND poniendo el resultado en la dirección $00 de ROM (no tiene efecto) para terminar de conseguir que los números sean aleatorios, haciendo avanzar la serie de números una posición cuando haya una tecla pulsada. Es totalmente imprevisible cuántas veces y durante cuánto tiempo se va a pulsar una tecla durante el juego.

A continuación, la pequeña rutina SACAFAN restablece en la pantalla los tres caracteres de los tampones de los fantasmas. Se emplea cuando hay que hacerlos desaparecer de la pantalla.

TECLADO

```
CD BB Ø2  TECLADO   CALL $Ø2BB
44                  LD B,H
4D                  LD C,L
51                  LD D,C
14                  INC D
C8                  RET Z
21 ØØ ØØ            LD HL,Ø
C5                  PUSH BC
CD 94 41            CALL RAND
C1                  POP BC
CD BD Ø7            CALL $Ø7BD
7E                  LD A,(HL)
21 4A 4Ø            LD HL,16458
FE 21               CP 33
2Ø Ø2               JR NZ NOIZDA3
36 Ø3               LD (HL),3
FE 22     NOIZDA3   CP 34
2Ø Ø2               JR NZ NOBAJA3
36 Ø1               LD (HL),1
FE 23     NOBAJA3   CP 35
2Ø Ø2               JR NZ NOSUBE3
36 ØØ               LD (HL),Ø
FE 24     NOSUBE3   CP 36
CØ                  RET NZ
36 Ø2               LD (HL),2
C9                  RET
21 3C 4Ø  SACAFAN   LD HL,16444
Ø6 Ø3               LD B,3
5E        OTRO3     LD E,(HL)
23                  INC HL
56                  LD D,(HL)
23                  INC HL
23                  INC HL
```



```
7E                  LD A,(HL)
12                  LD (DE),A
23                  INC HL
1Ø F6               DJNZ OTRO3
C9                  RET
```

Nos queda ahora el CONTROL DE FINALES, es decir, lo que hay que hacer cuando termina la acción (porque se ha terminado la comida o porque han atrapado al tipo) para dejar todo en condiciones de empezar de nuevo.

RSTCUADRO restablece la comida en el laberinto, cambiando todos los caracteres 128 (cuadro negro) por 155 (punto en negativo).

FINCUADRO actúa cuando la comida ha terminado. Pone la dirección del tipo en 0 (arriba), pone un cuadro negro en la posición donde estaba el tipo, saca los fantasmas, hace una pausa, restablece la comida y salta al lazo principal de acción.

MUERTO actúa cuando el tipo es atrapado, colocando un fantasma en su lugar; saca los fantasmas tras una pausa, decrementa el contador de tipos que quedan y si ha llegado a cero regresa al BASIC mediante la rutina de ROM de $0CDC, que da un mensaje de STOP. Si no pone el tipo y los fantasmas y salta al lazo de acción principal.

CONTROL DE FINALES

```
2A ØC 4Ø  RSTCUADRO  LD HL,(16396)
Ø6 EF                LD B,239
23        OTRO4      INC HL
7E                   LD A,(HL)
FE 8Ø                CP 128
2Ø Ø2                JR NZ SIGUE
36 9B                LD (HL),155
1Ø F6     SIGUE      DJNZ OTRO4
C9                   RET
AF        FINCUADRO  XOR A
32 4B 4Ø             LD (16459),A
2A 48 4Ø             LD HL,(16456)
```

```
36 8Ø                    LD (HL),128
CD 35 42                 CALL SACAFAN
CD 9C 42                 CALL PAUSE
CD 45 42                 CALL RSTCUADRO
18 25                    JR ACCION
2A 48 4Ø      MUERTO     LD HL,(16456)
36 ØB                    LD (HL),11
CD 9C 42                 CALL PAUSE
CD 35 42                 CALL SACAFAN
21 4C 4Ø      CUENTA     LD HL,1646Ø
35                       DEC (HL)
CA DC ØC                 JP Z $ØCDC ...Stop
CD CA 4Ø                 CALL PONE
18 ØE                    JR ACCION
```

Por último, está el lazo principal de desarrollo, que controla la secuencia de todo el juego. Al principio hay una llamada a una parte de la rutina «RAND» de ROM, que pone la variable de sistema del ZX81 «SEED» (que utilizamos para nuestros números aleatorios) según el tiempo que lleve conectado el ZX81 (como hace RAND en el BASIC). Luego pone a cero el contador de comida y a 4 el de tipos (1024 es 4×256+0), cuenta el primero y entra en el lazo de acción, donde se llama a las rutinas TECLADO, MTO TIPO, MTO FAN y LAPSO, por este orden. PAUSE produce lo que su nombre indica y LAPSO es un retardo que determina la velocidad del juego. Cambiando «LD BC,5000» por «LD BC,1» se verá la velocidad máxima del juego, que no permite jugar mucho tiempo.

LAZO PRINCIPAL DESARROLLO

```
CD 73 ØE                 CALL 3699
CD 92 4Ø                 CALL PONELAB
21 ØØ Ø4                 LD HL,1Ø24
22 4B 4Ø                 LD (16459),HL
18 E6                    JR CUENTA
CD Ø6 42      ACCION     CALL TECLADO
CD AA 41                 CALL MTOTIPO
```



```
CD 7B 41                 CALL MTOFAN
CD A5 42                 CALL LAPSO
18 F2                    JR ACCION
Ø1 ØØ ØØ      PAUSE      LD BC,ØØ
ØB            BUCLE      DEC BC
78                       LD A,B
B1                       OR C
2Ø FB                    JR NZ BUCLE
C9                       RET
Ø1 88 13      LAPSO      LD BC,5ØØØ
18 F5                    JR BUCLE
```



correr un programa

Para entrar el programa en el ZX81 necesitamos construir primero una línea «1 REM» con... 554 caracteres! Que no cunda el pánico porque no hay que teclear 554 veces (por otra parte no se puede). Utilizaremos la secuencia de sentencias del apartado 2.5 para unir líneas REM. Si se dispone de 16K el trabajo será más cómodo pero se puede hacer exactamente igual con 1K. Proceder como sigue:

1. Entrar a mano una línea «1 REM» con 110 caracteres cualesquiera, por ejemplo ceros.
2. Hacer «PRINT PEEK 16511 + 256 * PEEK 16512. Debe dar 110. Si da otra cosa, «GOTO» al paso 1.
3. Editar la línea, cambiar el número de línea por «2» y quitar con «DELETE» 5 (cinco) caracteres. De esta manera, al juntar las líneas juntamos también los cinco caracteres de número de línea, longitud y el propio «REM», así como el código 118 de fin de línea. Entrar la línea con «NEWLINE».

4. Editar la línea y cambiar su número por un «3». Entrar la línea.
5. Entrar la secuencia de sentencias para unir dos líneas REM dando números de línea a partir de 10, de diez en diez.
6. Poner el cursor en la línea «3 REM».
7. Entrar «RUN». No listar a partir de ahora.
8. Pulsar «EDIT». Aparecerá la línea «3 REM». Cambiar su número de línea por «2». Entrar la línea. Entrar «RUN».
9. Repetir lo mismo que en el paso 8.
10. Entrar «RUN».
11. Entrar las líneas 5 a 60 del cargador hexadecimal del apartado 2.7. Estas líneas deben anular las de la secuencia para unir REMs anterior.
12. Entrar como comando (sin número de línea) "PRINT PEEK 16511 + PEEK 16512 * 256". Si da 554 está todo bien; en caso contrario ha habido algún error. Volver a empezar.
13. Grabar en un cassette lo realizado un par de veces, para no tener que repetirlo si algo va mal de ahora en adelante.
14. Entrar «RUN». Entrar 8 ó 9 códigos y pulsar NEWLINE, y así sucesivamente hasta terminar. Empezar por la tabla del laberinto y a continuación los listados por el mismo orden en que aparecen en el texto y unidos unos a continuación de otros.
15. Entrar una línea "2 LET L = USR 17024". No anular esta línea en lo sucesivo bajo ningún concepto, porque se enganchará la línea «1REM», aunque se puede cambiar entrando otra en su lugar. Grabar unas copias en cassette.
16. Anular las líneas del cargador hexadecimal, sin tocar la línea «2 REM». Esta es la copia definitiva. Grabarla en cassette.

El programa se pone en marcha con «RUN» y no se para hasta que se acaba. Como es lógico sólo funciona correctamente con la estructura de 1K en el archivo de pantalla. Si se tiene conectados más de 3 1/4K se pueden anular desconectándolos o entrando "POKE 16389, 68», que corresponde al byte más significativo de la variable de sistema RAMTOP.

CAPÍTULO 19

DIRECCIONES UTILES DE LA ROM

La ROM del ZX81 contiene 8K de listados en CM y tablas que constituyen el intérprete BASIC. Es interesante disponer del listado de estos 8K al programar en CM en el ZX81, porque así se pueden aprovechar rutinas o trozos de ellas que residen en ROM y convengan a nuestros propósitos. No es el objeto de este libro dar este listado, pero por lo menos veremos una serie de direcciones de rutinas útiles de ROM indicando qué hacen y los requisitos en cuanto a registros que emplean y cómo utilizarlas. Esto nos evitará tener que salir al BASIC para poner al ZX81 en FAST, por ejemplo, dando la posibilidad de llamar directamente a la rutina de ROM que pone en FAST desde nuestro programa en CM.

19.1. Los «RST»:

La instrucción «RST n» hemos visto que es como «CALL» pero a determinadas direcciones de ROM (ver Cap. 13). Esto es lo que hace cada rutina:

RST $00: START. Tiene el mismo efecto que desconectar y volver a conectar el ZX81. Deja la memoria «limpia» como si lo hubiésemos desconectado.

RST $08: ERROR. El siguiente byte a «RST $08» en el listado debe ser un dato, porque la rutina lo interpretará como tal. Se produce un informe de error según este dato. Se pueden obtener todos los errores del BASIC y algunos otros:

Dato $00 a $0E da los informes de error del BASIC, de tal forma que el informe es uno más que el dato. Así dato 00 da error 1, $01 da error 2, etc.

Dato $0F a $22 da errores entre G y Z.

Dato $23 da error «espacio», $53 da «gráfico 1» y $54 da «gráfico A».

Dato $FF produce una vuelta al BASIC parecida a la de «RET» pero la función «USR» no da ningún resultado y la línea que la contiene no se ejecuta. Por ejemplo, si era «LET L = USR nn» no se asigna nada a la variable L, y se sigue con la siguiente línea del programa, si existe. Además siempre se vuelve al BASIC aunque se ejecute RST desde una subrutina. La pila («stack») se vacía automáticamente.

RST $10: PRINT. Efectúa un «PRINT» del carácter cuyo código esté en el registro A. Debe ser un código que corresponda a un carácter. Si el bit 1 de la variable de sistema FLAGS ($4001) es cero se ejecuta un «PRINT» y si es uno se ejecuta un «LPRINT» en la impresora, si está conectada, aunque en realidad coloca el código en el tampón de la impresora sin activarla necesariamente.

RST $18: COLLECT CHR. El contenido de la dirección de memoria indicada en la variable de sistema CH—ADD se carga en el registro A. Si es un espacio se ignora y sigue con RST $20.

RST $20: COLLECT NEXT CHAR. Igual que RST $18 pero CH—ADD se incrementa antes de recoger el carácter.

RST $28: FP—CALCULATOR. Pone en marcha el calculador en coma flotante. Se analiza con más detalle en el Cap. 20.

RST $30: MAKE BC SPACES. Crea espacio en la zona LINEA DE ENTRADA (ver mapa de memoria en Apéndice IV). Expande esta zona el número de bytes contenido en el par BC.

RST $38: INTERRUPT. Forma parte de la rutina de DISPLAY, de modo que más vale no tocarla.

19.2. Otras direcciones útiles

SAVE: Dirección $02F6. Corresponde al comando de este nombre. No funcionará directamente porque al principio busca el nombre del programa explorando la línea BASIC desde la que supone que le llama el intérprete. Se puede desensamblar la parte del principio para llamarla en otro punto y soslayar este problema.

LOAD: Dirección $0340. Corresponde al comando LOAD. Tampoco funciona por el mismo problema que SAVE, pero puede evitarse llamando a $0345 si el ZX81 está en FAST.

NEW: Dirección $03C3. Tiene el mismo efecto que el comando. Puede ser útil si se llama en programas que residan tras RAMTOP, en combinación con LOAD, por ejemplo.

COPY: Dirección $0869. Copia el contenido de la pantalla en la impresora. Si se llama a la dirección $086B copiará sólo las n primeras líneas si cargamos n en el registro D antes de llamar la rutina.

PRINT AT: Dirección $08F5. Posiciona las variables de sistema de acuerdo con las coordenadas de fila y columna, que deben estar en los registros B y C respectivamente antes de llamar a la rutina. El PRINT se puede ejecutar con un RST $10 después de llamar a «PRINT AT», por ejemplo.

HACER ESPACIO: Dirección $099D. Se expande la zona desplazando todo el contenido de RAM desde la dirección indicada por el par HL hasta la ZONA LIBRE una posición.

Si se llama la rutina en $099E en lugar de un espacio se crean BC espacios. La zona expandida no se limpia, es decir, contiene restos de lo que había antes de la expansión. Se ajustan automáticamente todas las variables de sistema que son punteros en RAM.

AJUSTA PUNTEROS: Dirección $09AD. Forma parte de la rutina anterior. Ajusta todas las variables de sistema que son punteros en RAM sumándoles la cantidad contenida en BC. Sólo se corrigen los punteros cuya dirección en la zona de variables es igual o mayor que la indicada por el par HL al empezar la rutina.

DIRECCION DE LINEA: Dirección $09F2. Encuentra la dirección en RAM de la siguiente línea BASIC o de la siguiente variable. En la entrada HL debe indicar el primer byte del número de línea o variable. La dirección de inicio de la siguiente línea o variable está a la vuelta en el par DE. Esta rutina es útil para explorar el listado BASIC en busca de líneas REM que deban interpretarse desde el CM, por ejemplo.

CLS: Dirección $0A2A. Tiene el mismo efecto que el comando «CLS». Si se llama en $0A2C se borran sólo las D últimas líneas, empezando a contar desde la 24, no desde la 22.

ANULA SITIO: Dirección $0A5D. Anula sitio en una parte de RAM

de modo inverso a como lo hace HACER SITIO, ajustando todos los punteros en la zona de variables de sistema. El par HL indica la dirección inicial de la zona a anular y el par DE la del final de la zona.

PLOT—UNPLOT: Una forma práctica de utilizar parte de la rutina «PLOT» en CM es la siguiente:

En B: coordenada Y. En C: coordenada X.

```
PUSH HL
PUSH DE
PUSH BC
LD HL,(16432)
PUSH HL
LD HL, $0C3E          (otro valor da UNPLOT)
LD (16432),HL
CALL $0BB2
POP HL
LD (16432),HL
POP BC
POP DE
POP HL
```

SCROLL: Dirección $0C0E.

RAND: La rutina empieza en $0E6C pero para usarla en código máquina hay que llamarla a $0E73. Esto no proporciona números aleatorios: sólo pone la variable de sistema SEED según esté FRAMES, contador que se decrementa 50 veces por segundo.

3292d. *STOP:* Dirección $0CDC. Vuelve al BASIC esté la pila («stack») como esté. Se utilizará como un salto en lugar de como subrutina.

TEST DE ESPACIO: Dirección $0EC5. Comprueba que entre el puntero STKEND y el de la pila («stack») del Z80 haya por lo menos 36 bytes. Si hay menos se detiene con error 4.

PONER EN FAST: Dirección $0F234.

PONER EN SLOW: Dirección $0F2B.

25360
25383

CAPÍTULO 20

CALCULOS MAS COMPLEJOS CON LA ROM

Cuando trabaja en BASIC, el ZX81 representa los números en una forma especial llamada «coma decimal flotante» («floating point» en inglés). Este tipo de representación permite números mayores que 65535, con decimales, y sus principios son bastante complejos y no hacen práctico su manejo en CM. Sin embargo, casi la mitad de la ROM está destinada a tratar este tipo de números. Hay montadas rutinas para calcular senos, raíces cuadradas, etc. (todas las funciones disponibles en BASIC). Para el manejo de estos números el ZX81 dispone de un calculador.

Aunque estos cálculos son algo más lentos que los que podamos realizar en CM con nuestras rutinas (sencillamente porque son más complejos), tenemos la posibilidad de utilizar el calculador para nuestros programas con buen rendimiento si utilizamos números entre 0 y 65535, los habituales en CM.

La representación de un número en coma flotante ocupa cinco bytes sea cual sea este número. El calculador trabaja con una pila («stack») especial, distinta a la pila del Z80: es la pila («stack») del calculador (ver mapa de memoria en Apéndice IV). Se diferencia de ésta en que cada ítem de la pila («stack») son cinco bytes en lugar de dos y en que las instrucciones para manejarlo son algo más potentes que las de la pila («stack») del Z80.

Estas instrucciones son en realidad otras subrutinas de ROM que se llaman con CASLL. Son las siguientes:



llamadas

CALL STACK—A: Dirección $151D. Es una especie de equivalente de «PUSH A» de CM (si existiera). Coloca en la cima de la pila («stack») del calculador el dato contenido en el registro A, una vez transformado en la extraña forma de cinco bytes que utiliza el calculador. Con esto sólo podemos guardar números entre $00 y $FF, como es lógico.

CALL STACK—BC: Dirección $1520. Hace lo mismo que «CALL STACK—A» pero con el par de registros BC, lo que permite guardar números entre 0 y 65535.

CALL FP—TO—A: Dirección $15CD. Recupera el número de la cima de la pila («stack») del calculador transformándolo en binario y cargándolo en el registro A. El número de la cima de la pila («stack») debe ser igual o menor que 255, es decir, debe caber en el registro A. Es el equivalente del «POP» de CM.

CALL FP—TO—BC: Dirección $0EA7. Trabaja igual que «CALL FP—TO—A» pero el resultado se carga en el par BC. El número debe estar entre 0 y 65535.

Para operar dos números en esta forma, primero hay que ponerlos en la pila («stack») del calculador con las rutinas explicadas y entonces usar la instrucción «RST $28», que activa el calculador. Los bytes colocados a continuación del código de «RST $28» ($EF) se interpretan por esta rutina como datos que indican las operaciones que debe efectuar hasta que encuentra el dato $34 que tiene el significado de «fin de los cálculos». A partir de ahí se interpreta el CM normalmente.

Los códigos para las cuatro operaciones básicas son:

$0F para la suma
$03 para la resta
$04 para la multiplicación
$05 para la división

Cada una de estas operaciones saca los dos últimos números de la cima de la pila («stack») del calculador, efectúa la operación y deja en la cima de la pila («stack») otra vez el resultado, habiendo desaparecido de ahí los operandos.

Por ejemplo para multiplicar $05 por $07:

```
INICIO   LD A,$05
         CALL STACK—A      guarda A en la pila («stack») del
                           calculador
         LD A,$07
         CALL STACK—A      guarda A en la pila («stack») del
                           calculador
         RST $28           activa el calculador
         DEFB $04          código de multiplicación
         DEFB $34          fin de los cálculos
         CALL FP—TO—A      recupera el resultado en A
         RET               fin
```

DEFB es una especie de mnemónico que no corresponde a ninguna instrucción. Se emplea en los listados de CM para indicar que el código es un dato.

Se puede hacer varias operaciones seguidas siempre que los datos estén en la pila («stack»), empleando el resultado de una operación como operando de la siguiente. Para multiplicar $03 por $04 y sumarle $09 al resultado:

```
INICIO   LD A,$03
         CALL STACK—A
         LD A,$04
         CALL STACK—A
         LD A,$09
         CALL STACK—A
         RST $28
         DEFB $04
         DEFB $0F
         DEFB $34
         CALL FP—TO—A
         RET
```

Pero se puede hacer mucho más que las cuatro operaciones básicas con el calculador del ZX81. De la misma forma, se puede emplear cualquier función de las disponibles en BASIC. Su código se obtiene restando $AB (171 dec.) del código de la función, que se puede encontrar en el apéndice A del Manual de Sinclair del ZX81. Algunas de las más útiles:

$1C para SIN
$25 para SQR
$22 para LN
$27 para ABS
$28 para PEEK
$29 para USR
$2A para STR$

Algunas de éstas corresponden a funciones que implican el uso de cadenas, como STR$. La pila («stack») del calculador puede manejar cadenas alfanuméricas además de números. Por ejemplo STR$ convierte el número de la cima de la pila («stack») en su representación como cadena (es decir en los códigos de sus dígitos en decimal). Esto puede ser interesante para convertir números de binario a una forma que se puede trasladar directamente a la pantalla para obtener la presentación del número. Para ello hay que trasladar los códigos directamente de la cima de la pila («stack»), sin utilizar CALL FP—TO—BC, empleando como referencia el puntero de la pila («stack») del calculador STKEND, que es una variable de sistema.

Estas funciones toman el número de la cima de la pila («stack») y dejan el resultado en el lugar de donde han tomado el número. Por ejemplo PEEK traslada el contenido de la dirección de memoria indicada por el número de la cima de la pila («stack») a esa misma cima, transformándolo previamente en la forma de cinco bytes de la representación en coma flotante. USR salta a la dirección indicada por el número de la cima y sigue desde ahí en CM normal (como hace USR al emplearla desde el BASIC) pero al regresar sigue con los cálculos que haya después del código de USR.

Disponemos además de seis memorias para guardar resultados parciales de la cima de la pila («stack»). Funcionan como las memorias de las calculadoras, y siempre se toma o deja el dato de la cima de la pila («stack»), en representación de 5 bytes. Para emplearlas se utilizan códigos como para el resto de las operaciones con el calculador, introducidos en el listado de «DEFBs» en los lugares adecuados. Los códigos son:

$C0: guarda el número en la memoria 0
$C1: guarda el número en la memoria 1
$C2: ídem. en la memoria 2
$C3: ídem. en la memoria 3
$C4: ídem en la memoria 4
$C5: ídem en la memoria 5
$E0: recupera el número guardado en la memoria 0
$E1: ídem. para la memoria 1
$E2: ídem. para la memoria 2
$E3: ídem. para la memoria 3
$E4: ídem. para la memoria 4
$E5: ídem. para la memoria 5

El hecho de guardar un número en una memoria no lo elimina de la pila («stack»).

Unas últimas consideraciones:

CALL $13F8 tiene el efecto de eliminar el número de la cima de la pila («stack») y cargarlo tal como está, en la notación de cinco bytes, en los registros A,E,D,C, y B, en este orden.

CALL $12C3 tiene el efecto inverso, guardando en la pila («stack») los registros A,E,D,C y B, por este orden sin ninguna modificación.

El código $01 intercambia las posiciones de los dos últimos números de la cima de la pila («stack»), en su forma de cinco bytes.

APENDICE I
INSTRUCCIONES DEL Z80

(El valor atribuido a *d* en el Código Objeto es 05)

Código Objeto	Código Fuente	
8E	ADC	A,(HL)
DD8E05	ADC	A,(IX+d)
FD8E05	ADC	A,(IY+d)
8F	ADC	A,A
88	ADC	A,B
89	ADC	A,C
8A	ADC	A,D
8B	ADC	A,E
8C	ADC	A,H
8D	ADC	A,L
CE20	ADC	A,n
ED4A	ADC	HL,BC
ED5A	ADC	HL,DE
ED6A	ADC	HL,HL
ED7A	ADC	HL,SP
86	ADD	A,(HL)
DD8605	ADD	A,(IX+d)
FD8605	ADD	A,(IY+d)
87	ADD	A,A
80	ADD	A,B
81	ADD	A,C
82	ADD	A,D
83	ADD	A,E
84	ADD	A,H
85	ADD	A,L
C620	ADD	A,n
09	ADD	HL,BC
19	ADD	HL,DE
29	ADD	HL,HL
39	ADD	HL,SP
DD09	ADD	IX,BC
DD19	ADD	IX,DE
DD29	ADD	IX,IX
DD39	ADD	IX,SP
FD09	ADD	IY,BC
FD19	ADD	IY,DE
FD29	ADD	IY,IY
FD39	ADD	IY,SP
A6	AND	(HL)
DDA605	AND	(IX+d)
FDA605	AND	(IY+d)
A7	AND	A
A0	AND	B
A1	AND	C
A2	AND	D
A3	AND	E
A4	AND	H
A5	AND	L

Código Objeto	Código Fuente	
E620	AND	n
CB46	BIT	0,(HL)
DDCB0546	BIT	0,(IX+d)
FDCB0546	BIT	0,(IY+d)
CB47	BIT	0,A
CB40	BIT	0,B
CB41	BIT	0,C
CB42	BIT	0,D
CB43	BIT	0,E
CB44	BIT	0,H
CB45	BIT	0,L
CB4E	BIT	1 (HL)
DDCB054E	BIT	1,(IX+d)
FDCB054E	BIT	1,(IY+d)
CB4F	BIT	1,A
CB48	BIT	1,B
CB49	BIT	1,C
CB4A	BIT	1,D
CB4B	BIT	1,E
CB4C	BIT	1,H
CB4D	BIT	1,L
CB56	BIT	2,(HL)
DDCB0556	BIT	2,(IX+d)
FDCB0556	BIT	2,(IY+d)
CB57	BIT	2,A
CB50	BIT	2,B
CB51	BIT	2,C
CB52	BIT	2,D
CB53	BIT	2,E
CB54	BIT	2,H
CB55	BIT	2,L
CB5E	BIT	3,(HL)
DDCB055E	BIT	3,(IX+d)
FDCB055E	BIT	3,(IY+d)
CB5F	BIT	3,A
CB58	BIT	3,B
CB59	BIT	3,C
CB5A	BIT	3,D
CB5B	BIT	3,E
CB5C	BIT	3,H
CB5D	BIT	3,L
CB66	BIT	4,(HL)
DDCB0566	BIT	4,(IX+d)
FDCB0566	BIT	4,(IY+d)
CB67	BIT	4,A
CB60	BIT	4,B
CB61	BIT	4,C
CB62	BIT	4,D

Código Objeto	Código Fuente	
CB63	BIT	4,E
CB64	BIT	4,H
CB65	BIT	4,L
CB6E	BIT	5,(HL)
DDCB056E	BIT	5,(IX+d)
FDCB056E	BIT	5,(IY+d)
CB6F	BIT	5,A
CB68	BIT	5,B
CB69	BIT	5,C
CB6A	BIT	5,D
CB6B	BIT	5,E
CB6C	BIT	5,H
CB6D	BIT	5,L
CB76	BIT	6,(HL)
DDCB0576	BIT	6,(IX+d)
FDCB0576	BIT	6,(IY+d)
CB77	BIT	6,A
CB70	BIT	6,B
CB71	BIT	6,C
CB72	BIT	6,D
CB73	BIT	6,E
CB74	BIT	6,H
CB75	BIT	6,L
CB7E	BIT	7,(HL)
DDCB057E	BIT	7,(IX+d)
FDCB057E	BIT	7,(IY+d)
CB7F	BIT	7,A
CB78	BIT	7,B
CB79	BIT	7,C
CB7A	BIT	7,D
CB7B	BIT	7,E
CB7C	BIT	7,H
CB7D	BIT	7,L
DC8405	CALL	C,nn
FC8405	CALL	M,nn
D48405	CALL	NC,nn
C48405	CALL	NZ,nn
F48405	CALL	P,nn
EC8405	CALL	PE,nn
E48405	CALL	PO,nn
CC8405	CALL	Z,nn
CD8405	CALL	nn
3F	CCF	
BE	CP	(HL)
DDBE05	CP	(IX+d)
FDBE05	CP	(IY+d)
BF	CP	A
B8	CP	B
B9	CP	C
BA	CP	D
BB	CP	E
BC	CP	H
BD	CP	L
FE20	CP	n
EDA9	CPD	
EDB9	CPDR	

Código Objeto	Código Fuente	
EDB1	CPIR	
EDA1	CPI	
2F	CPL	
27	DAA	
35	DEC	(HL)
DD3505	DEC	(IX+d)
FD3505	DEC	(IY+d)
3D	DEC	A
05	DEC	B
0B	DEC	BC
0D	DEC	C
15	DEC	D
1B	DEC	DE
1D	DEC	E
25	DEC	H
2B	DEC	HL
DD2B	DEC	IX
FD2B	DEC	IY
2D	DEC	L
3B	DEC	SP
F3	DI	
102E	DJNZ	e
FB	EI	
E3	EX	(SP),HL
DDE3	EX	(SP),IX
FDE3	EX	(SP),IY
08	EX	AF,AF'
EB	EX	DE,HL
D9	EXX	
76	HALT	
ED46	IM	0
ED56	IM	1
ED5E	IM	2
ED78	IN	A,(C)
ED40	IN	B,(C)
ED48	IN	C,(C)
ED50	IN	D,(C)
ED58	IN	E,(C)
ED60	IN	H,(C)
ED68	IN	L,(C)
34	INC	(HL)
DD3405	INC	(IX+d)
FD3405	INC	(IY+d)
3C	INC	A
04	INC	B
03	INC	BC
0C	INC	C
14	INC	D
13	INC	DE
1C	INC	E
24	INC	H
23	INC	HL
DD23	INC	IX
FD23	INC	IY
2C	INC	L
33	INC	SP
DB20	IN	A,(n)

Código Objeto	Código Fuente	
EDAA	IND	
EDBA	INDR	
EDA2	INI	
EDB2	INIR	
C38405	JP	nn
E9	JP	(HL)
DDE9	JP	(IX)
FDE9	JP	(IY)
DA8405	JP	C,nn
FA8405	JP	M,nn
D28405	JP	NC,nn
C28405	JP	NZ,nn
F28405	JP	P,nn
EA8405	JP	PE,nn
E28405	JP	PO,nn
CA8405	JP	Z,nn
382E	JR	C,e
302E	JR	NC,e
202E	JR	NZ,e
282E	JR	Z,e
182E	JR	e
02	LD	(BC),A
12	LD	(DE),A
77	LD	(HL),A
70	LD	(HL),B
71	LD	(HL),C
72	LD	(HL),D
73	LD	(HL),E
74	LD	(HL),H
75	LD	(HL),L
3620	LD	(HL),n
DD7705	LD	(IX+d),A
DD7005	LD	(IX+d),B
DD7105	LD	(IX+d),C
DD7205	LD	(IX+d),D
DD7305	LD	(IX+d),E
DD7405	LD	(IX+d),H
DD7505	LD	(IX+d),L
DD360520	LD	(IX+d),n
FD7705	LD	(IY+d),A
FD7005	LD	(IY+d),B
FD7105	LD	(IY+d),C
FD7205	LD	(IY+d),D
FD7305	LD	(IY+d),E
FD7405	LD	(IY+d),H
FD7505	LD	(IY+d),L
FD360520	LD	(IY+d),n
328405	LD	(nn),A
ED438405	LD	(nn),BC
ED538405	LD	(nn),DE
228405	LD	(nn),HL
DD228405	LD	(nn),IX
FD228405	LD	(nn),IY
ED738405	LD	(nn),SP
0A	LD	A,(BC)
1A	LD	A,(DE)
7E	LD	A,(HL)

Código Objeto	Código Fuente	
DD7E05	LD	A,(IX+d)
FD7E05	LD	A,(IY+d)
3A8405	LD	A,(nn)
7F	LD	A,A
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
ED57	LD	A,I
7D	LD	A,L
3E20	LD	A,n
ED5F	LD	A,R
46	LD	B,(HL)
DD4605	LD	B,(IX+d)
FD4605	LD	B,(IY+d)
47	LD	B,A
40	LD	B,B
41	LD	B,C
42	LD	B,D
43	LD	B,E
44	LD	B,H
45	LD	B,L
0620	LD	B,n
ED4B8405	LD	BC,(nn)
018405	LD	BC,nn
4E	LD	C,(HL)
DD4E05	LD	C,(IX+d)
FD4E05	LD	C,(IY+d)
4F	LD	C,A
48	LD	C,B
49	LD	C,C
4A	LD	C,D
4B	LD	C,E
4C	LD	C,H
4D	LD	C,L
0E20	LD	C,n
56	LD	D,(HL)
DD5605	LD	D,(IX+d)
FD5605	LD	D,(IY+d)
57	LD	D,A
50	LD	D,B
51	LD	D,C
52	LD	D,D
53	LD	D,E
54	LD	D,H
55	LD	D,L
1620	LD	D,n
ED5B8405	LD	DE,(nn)
118405	LD	DE,nn
5E	LD	E,(HL)
DD5E05	LD	E,(IX+d)
FD5E05	LD	E,(IY+d)
5F	LD	E,A
58	LD	E,B
59	LD	E,C
5A	LD	E,D

Código Objeto	Código Fuente	
5B	LD	E,E
5C	LD	E,H
5D	LD	E,L
1E20	LD	E,n
66	LD	H,(HL)
DD6605	LD	H,(IX+d)
FD6605	LD	H,(IY+d)
67	LD	H,A
60	LD	H,B
61	LD	H,C
62	LD	H,D
63	LD	H,E
64	LD	H,H
65	LD	H,L
2620	LD	H,n
2A8405	LD	HL,(nn)
218405	LD	HL,nn
ED47	LD	I,A
DD2A8405	LD	IX,(nn)
DD218405	LD	IX,nn
FD2A8405	LD	IY,(nn)
FD218405	LD	IY,nn
6E	LD	L,(HL)
DD6E05	LD	L,(IX+d)
FD6E05	LD	L,(IY+d)
6F	LD	L,A
68	LD	L,B
69	LD	L,C
6A	LD	L,D
6B	LD	L,E
6C	LD	L,H
6D	LD	L,L
2E20	LD	L,n
ED4F	LD	R,A
ED7B8405	LD	SP,(nn)
F9	LD	SP,HL
DDF9	LD	SP,IX
FDF9	LD	SP,IY
318405	LD	SP,nn
EDA8	LDD	
EDB8	LDDR	
EDA0	LDI	
EDB0	LDIR	
ED44	NEG	
00	NOP	
B6	OR	(HL)
DDB605	OR	(IX+d)
FDB605	OR	(IY+d)
B7	OR	A
B0	OR	B
B1	OR	C
B2	OR	D
B3	OR	E
B4	OR	H
B5	OR	L
F620	OR	n
EDBB	OTDR	

Código Objeto	Código Fuente	
EDB3	OTIR	
ED79	OUT	(C),A
ED41	OUT	(C),B
ED49	OUT	(C),C
ED51	OUT	(C),D
ED59	OUT	(C),E
ED61	OUT	(C),H
ED69	OUT	(C),L
D320	OUT	(n),A
EDAB	OUTD	
EDA3	OUTI	
F1	POP	AF
C1	POP	BC
D1	POP	DE
E1	POP	HL
DDE1	POP	IX
FDE1	POP	IY
F5	PUSH	AF
C5	PUSH	BC
D5	PUSH	DE
E5	PUSH	HL
DDE5	PUSH	IX
FDE5	PUSH	IY
CB86	RES	0,(HL)
DDCB0586	RES	0,(IX+d)
FDCB0586	RES	0,(IY+d)
CB87	RES	0,A
CB80	RES	0,B
CB81	RES	0,C
CB82	RES	0,D
CB83	RES	0,E
CB84	RES	0,H
CB85	RES	0,L
CB8E	RES	1,(HL)
DDCB058E	RES	1,(IX+d)
FDCB058E	RES	1,(IY+d)
CB8F	RES	1,A
CB88	RES	1,B
CB89	RES	1,C
CB8A	RES	1,D
CB8B	RES	1,E
CB8C	RES	1,H
CB8D	RES	1,L
CB96	RES	2,(HL)
DDCB0596	RES	2,(IX+d)
FDCB0596	RES	2,(IY+d)
CB97	RES	2,A
CB90	RES	2,B
CB91	RES	2,C
CB92	RES	2,D
CB93	RES	2,E
CB94	RES	2,H
CB95	RES	2,L
CB9E	RES	3,(HL)
DDCB059E	RES	3,(IX+d)
FDCB059E	RES	3,(IY+d)

Código Objeto	Código Fuente	
CB9F	RES	3,A
CB98	RES	3,B
CB99	RES	3,C
CB9A	RES	3,D
CB9B	RES	3,E
CB9C	RES	3,H
CB9D	RES	3,L
CBA6	RES	4,(HL)
DDCB05A6	RES	4,(IX+d)
FDCB05A6	RES	4,(IY+d)
CBA7	RES	4,A
CBA0	RES	4,B
CBA1	RES	4,C
CBA2	RES	4,D
CBA3	RES	4,E
CBA4	RES	4,H
CBA5	RES	4,L
CBAE	RES	5,(HL)
DDCB05AE	RES	5,(IX+d)
FDCB05AE	RES	5,(IY+d)
CBAF	RES	5,A
CBA8	RES	5,B
CBA9	RES	5,C
CBAA	RES	5,D
CBAB	RES	5,E
CBAC	RES	5,H
CBAD	RES	5,L
CBB6	RES	6,(HL)
DDCB05B6	RES	6,(IX+d)
FDCB05B6	RES	6,(IY+d)
CBB7	RES	6,A
CBB0	RES	6,B
CBB1	RES	6,C
CBB2	RES	6,D
CBB3	RES	6,E
CBB4	RES	6,H
CBB5	RES	6,L
CBBE	RES	7,(HL)
DDCB05BE	RES	7,(IX+d)
FDCB05BE	RES	7,(IY+d)
CBBF	RES	7,A
CBB8	RES	7,B
CBB9	RES	7,C
CBBA	RES	7,D
CBBB	RES	7,E
CBBC	RES	7,H
CBBD	RES	7,L
C9	RET	
D8	RET	C
F8	RET	M
D0	RET	NC
C0	RET	NZ
F0	RET	P
E8	RET	PE
E0	RET	PO
C8	RET	Z

Código Objeto	Código Fuente	
ED4D	RETI	
ED45	RETN	
CB16	RL	(HL)
DDCB0516	RL	(IX+d)
FDCB0516	RL	(IY+d)
CB17	RL	A
CB10	RL	B
CB11	RL	C
CB12	RL	D
CB13	RL	E
CB14	RL	H
CB15	RL	L
17	RLA	
CB06	RLC	(HL)
DDCB0506	RLC	(IX+d)
FDCB0506	RLC	(IY+d)
CB07	RLC	A
CB00	RLC	B
CB01	RLC	C
CB02	RLC	D
CB03	RLC	E
CB04	RLC	H
CB05	RLC	L
07	RLCA	
ED6F	RLD	
CB1E	RR	(HL)
DDCB051E	RR	(IX+d)
FDCB051E	RR	(IY+d)
CB1F	RR	A
CB18	RR	B
CB19	RR	C
CB1A	RR	D
CB1B	RR	E
CB1C	RR	H
CB1D	RR	L
1F	RRA	
CB0E	RRC	(HL)
DDCB050E	RRC	(IX+d)
FDCB050E	RRC	(IY+d)
CB0F	RRC	A
CB08	RRC	B
CB09	RRC	C
CB0A	RRC	D
CB0B	RRC	E
CB0C	RRC	H
CB0D	RRC	L
OF	RRCA	
ED67	RRD	
C7	RST	00H
CF	RST	08H
D7	RST	10H
DF	RST	18H
E7	RST	20H
EF	RST	28H
F7	RST	30H
FF	RST	38H
DE20	SBC	A,n

Código Objeto	Código Fuente	
9E	SBC	A,(HL)
DD9E05	SBC	A,(IX+d)
FD9E05	SBC	A,(IY+d)
9F	SBC	A,A
98	SBC	A,B
99	SBC	A,C
9A	SBC	A,D
9B	SBC	A,E
9C	SBC	A,H
9D	SBC	A,L
ED42	SBC	HL,BC
ED52	SBC	HL,DE
ED62	SBC	HL,HL
ED72	SBC	HL,SP
37	SCF	
CBC6	SET	0,(HL)
DDCB05C6	SET	0,(IX+d)
FDCB05C6	SET	0,(IY+d)
CBC7	SET	0,A
CBC0	SET	0,B
CBC1	SET	0,C
CBC2	SET	0,D
CBC3	SET	0,E
CBC4	SET	0,H
CBC5	SET	0,L
CBCE	SET	1,(HL)
DDCB05CE	SET	1,(IX+d)
FDCB05CE	SET	1,(IY+d)
CBCF	SET	1,A
CBC8	SET	1,B
CBC9	SET	1,C
CBCA	SET	1,D
CBCB	SET	1,E
CBCC	SET	1,H
CBCD	SET	1,L
CBD6	SET	2,(HL)
DDCB05D6	SET	2,(IX+d)
FDCB05D6	SET	2,(IY+d)
CBD7	SET	2,A
CBD0	SET	2,B
CBD1	SET	2,C
CBD2	SET	2,D
CBD3	SET	2,E
CBD4	SET	2,H
CBD5	SET	2,L
CBD8	SET	3,B
CBDE	SET	3,(HL)
DDCB05DE	SET	3,(IX+d)
FDCB05DE	SET	3,(IY+d)
CBDF	SET	3,A
CBD9	SET	3,C
CBDA	SET	3,D
CBDB	SET	3,E
CBDC	SET	3,H
CBDD	SET	3,L
CBE6	SET	4,(HL)

Código Objeto	Código Fuente	
DDCB05E6	SET	4,(IX+d)
FDCB05E6	SET	4,(IY+d)
CBE7	SET	4,A
CBE0	SET	4,B
CBE1	SET	4,C
CBE2	SET	4,D
CBE3	SET	4,E
CBE4	SET	4,H
CBE5	SET	4,L
CBEE	SET	5,(HL)
DDCB05EE	SET	5,(IX+d)
FDCB05EE	SET	5,(IY+d)
CBEF	SET	5,A
CBE8	SET	5,B
CBE9	SET	5,C
CBEA	SET	5,D
CBEB	SET	5,E
CBEC	SET	5,H
CBED	SET	5,L
CBF6	SET	6,(HL)
DDCB05F6	SET	6,(IX+d)
FDCB05F6	SET	6,(IY+d)
CBF7	SET	6,A
CBF0	SET	6,B
CBF1	SET	6,C
CBF2	SET	6,D
CBF3	SET	6,E
CBF4	SET	6,H
CBF5	SET	6,L
CBFE	SET	7,(HL)
DDCB05FE	SET	7,(IX+d)
FDCB05FE	SET	7,(IY+d)
CBFF	SET	7,A
CBF8	SET	7,B
CBF9	SET	7,C
CBFA	SET	7,D
CBFB	SET	7,E
CBFC	SET	7,H
CBFD	SET	7,L
CB26	SLA	(HL)
DDCB0526	SLA	(IX+d)
FDCB0526	SLA	(IY+d)
CB27	SLA	A
CB20	SLA	B
CB21	SLA	C
CB22	SLA	D
CB23	SLA	E
CB24	SLA	H
CB25	SLA	L
CB2E	SRA	(HL)
DDCB052E	SRA	(IX+d)
FDCB052E	SRA	(IY+d)
CB2F	SRA	A
CB28	SRA	B
CB29	SRA	C
CB2A	SRA	D

Código Objeto	Código Fuente	
CB2B	SRA	E
CB2C	SRA	H
CB2D	SRA	L
CB3E	SRL	(HL)
DDCB053E	SRL	(IX+d)
FDCB053E	SRL	(IY+d)
CB3F	SRL	A
CB38	SRL	B
CB39	SRL	C
CB3A	SRL	D
CB3B	SRL	E
CB3C	SRL	H
CB3D	SRL	L
96	SUB	(HL)
DD9605	SUB	(IX+d)
FD9605	SUB	(IY+d)
97	SUB	A
90	SUB	B
91	SUB	C
92	SUB	D
93	SUB	E
94	SUB	H
95	SUB	L
D620	SUB	n
AE	XOR	(HL)
DDAE05	XOR	(IX+d)
FDAE05	XOR	(IY+d)
AF	XOR	A
A8	XOR	B
A9	XOR	C
AA	XOR	D
AB	XOR	E
AC	XOR	H
AD	XOR	L
EE20	XOR	n

APENDICE II

TABLA DE ALTERACIONES DE FLAGS

EFECTOS SOBRE LOS FLAGS

Instrucciones	S	Z	H	P	N	C
ADC A,r	x	x	x	x	0	x
ADC HL,ss	x	x	x	x	0	x
ADD A,r	x	x	x	x	0	x
ADD HL,ss	—	—	x	—	0	x
ADD IX,ss	—	—	x	—	0	x
ADD IY,ss	—	—	x	—	0	x
AND r	x	x	1	x	0	0
BIT b,r	?	x	1	?	0	—
CALL pq	—	—	—	—	—	—
CALL c,pq	—	—	—	—	—	—
CCF	—	—	!	—	0	x
(! toma el valor previo del flag C)						
CP r	x	x	x	x	1	x
CPI	x	!	x	!	1	—
CPD	x	!	x	!	1	—
CPIR	x	!	x	!	1	—
CPDR	x	!	x	!	1	—
!:Z es 1 si BC llega a 0 y P/V es 1 si A = (HL).						
CPL	—	—	1	—	1	—
DAA	x	x	x	x	—	x
DEC r	x	x	x	x	1	—
DEC ss	—	—	—	—	—	—
DI	—	—	—	—	—	—
DJNZ e	—	—	—	—	—	—
EI	—	—	—	—	—	—
EX AF,A'F'	—	—	—	—	—	—
EX DE,HL	—	—	—	—	—	—
EX (SP),HL	—	—	—	—	—	—
EX (SP),IX	—	—	—	—	—	—
EX (SP),IY	—	—	—	—	—	—
EXX	—	—	—	—	—	—
HALT	—	—	—	—	—	—
IM 0	—	—	—	—	—	—
IM 1	—	—	—	—	—	—
IM 2	—	—	—	—	—	—

Instrucciones	S	Z	H	P	N	C
INC r	x	x	x	x	0	—
INC ss	—	—	—	—	—	—
IN A,(n)	—	—	—	—	—	—
IN r,(C)	x	x	x	x	0	—
INI	?	x	?	?	1	—
IND	?	!	?	?	1	—
!:Z = 1 si B llega a 0						
INIR	?	1	?	?	1	—
INDR	?	1	?	?	1	—
JP ninguna instrucción altera ningún flag						
JR ninguna instrucción altera ningún flag						
LD (BC),A	—	—	—	—	—	—
LD A,(BC)	—	—	—	—	—	—
LD A,I	x	x	0	!	0	—
LD A,R	x	x	0	!	0	—
LD ninguna de las otras instrucciones LD altera ningún flag						
LDI	—	—	0	!	0	—
LDD	—	—	0	!	0	—
!:P/V = 0 si BC = 0						
LDIR	—	—	0	0	0	—
LDDR	—	—	0	0	0	—
NEG	x	x	x	x	1	x
NOP	—	—	—	—	—	—
OR r	x	x	0	x	0	0
OUT (n),A	—	—	—	—	—	—
OUT (C),r	—	—	—	—	—	—
OUTI	?	!	?	?	1	—
OUTD	?	!	?	?	1	—
!: Z = 0 si BC = 0						
OTIR	?	1	?	?	1	—
OTDR	?	1	?	?	1	—
POP AF	!	!	!	!	!	!

Instrucciones	Flags S	Z	H	P	N	C
!: los flags quedan determinados por el byte que viene del stack.						
POP ss	—	—	—	—	—	—
PUSH AF	—	—	—	—	—	—
PUSH ss	—	—	—	—	—	—
RES b,r	—	—	—	—	—	—
RET	—	—	—	—	—	—
RET c	—	—	—	—	—	—
RETN	—	—	—	—	—	—
RETI	—	—	—	—	—	—
RLA	—	—	0	—	0	x
RL r	x	x	0	x	0	x
RLCA	—	—	0	—	0	x
RLC r	x	x	0	x	0	x
RLD	x	x	0	x	0	—
RRA	—	—	0	—	0	x

Instrucciones	Flags S	Z	H	P	N	C
RR r	x	x	0	x	0	x
RRCA	—	—	0	—	0	x
RRC r	x	x	0	x	0	x
RRD	x	x	0	x	0	—
RST ninguna instrucción altera ningún flag						
SBC A,r	x	x	x	x	1	x
SBC HL,ss	x	x	x	x	1	x
SCF	—	—	0	—	0	1
SET b,r	—	—	—	—	—	—
SLA r	x	x	0	x	0	x
SRA r	x	x	0	x	0	x
SRL r	x	x	0	x	0	x
SUB r	x	x	x	x	1	x
XOR r	x	x	0	x	0	0

simbolos:

x: el flag es alterado por la instrucción según el contenido de los registros implicados.
—: el flag no se altera por esta instrucción en ningún caso.
1: el flag se pone a uno en cualquier caso.
0: el flag se pone a cero en cualquier caso.
?: el flag se pone a uno o a cero al azar.
!: comportamiento particular. Se da una explicación en cada caso.

FLAGS:

S: signo (en complemento a dos)
Z: cero
H: «halfcarry». Si hay carry entre el bit 3 y 4.
P: parity/overflow
N: sustracción
C: carry o acarreo.

Otros:

r: registro
ss: par de registros
e: desplazamiento
n: número entre 0 y 255
b: número entre 0 y 7
c: condición
pq: dirección

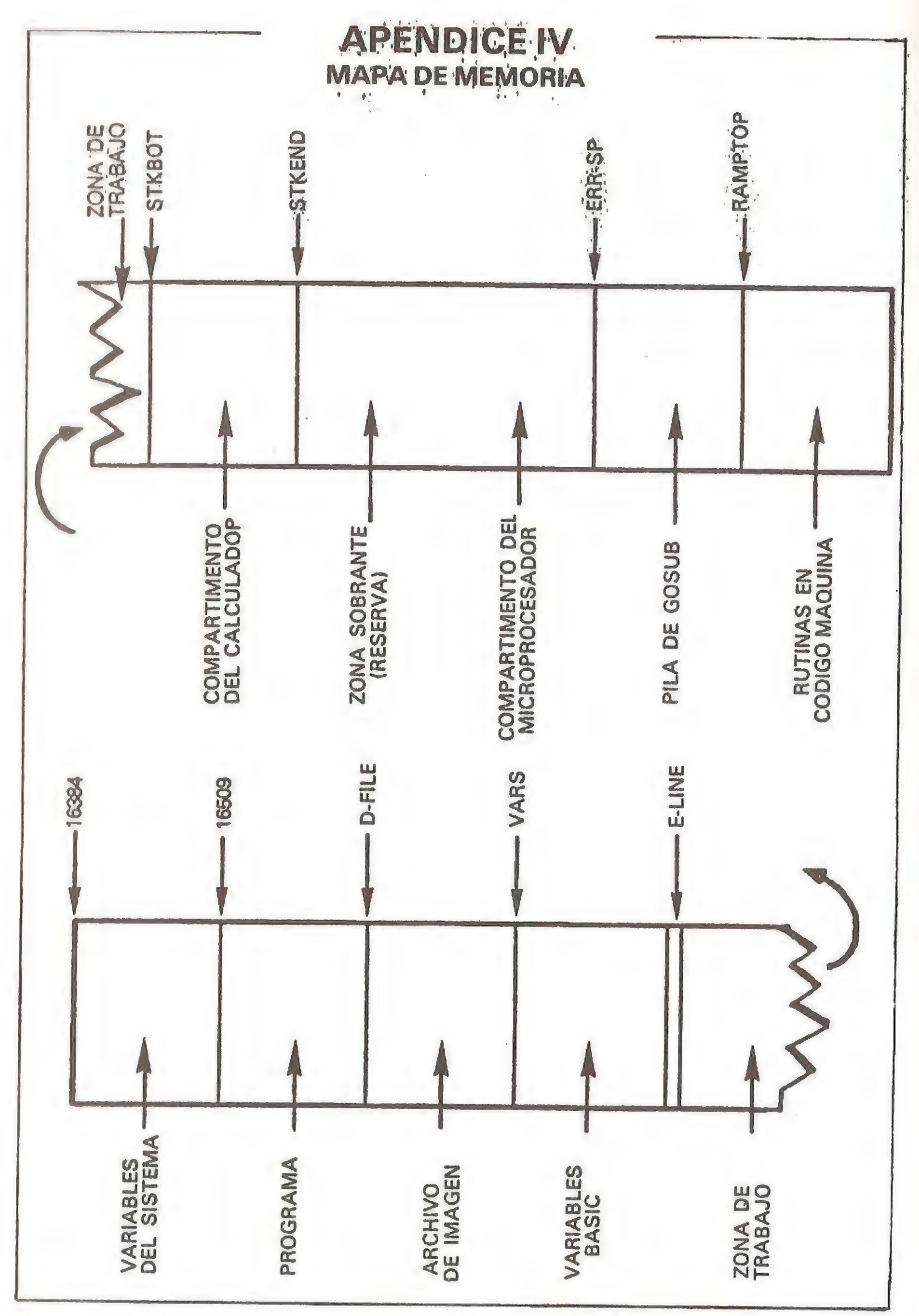
APENDICE III

TABLAS DE SALTOS RELATIVOS HACIA ADELANTE

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

TABLAS DE SALTOS RELATIVOS HACIA ATRAS

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1



APENDICE V

V.1. La estructura de pantalla en el ZX SPECTRUM

En el Spectrum, el Z80 no interviene a la hora de enviar la información del archivo de pantalla al modulador UHF, de modo que no hay ninguna rutina tipo DISPLAY que nos interrumpa nuestro programa como ocurre en el ZX81. Por otra parte, el Spectrum dispone de alta resolución (255 × 175) y se necesita poder determinar si cada punto de la pantalla es blanco o negro, independientemente de cualquier otro.

Debido a todo esto, el archivo de pantalla del Spectrum no tiene nada que ver con el ZX81.

En el ZX Spectrum este archivo de pantalla ocupa 6144 bytes entre las direcciones de memoria 16384 y 22527 inclusive. A diferencia de lo que ocurre en el ZX81, estas direcciones son fijas y siempre las mismas. Considerando el formato normal de pantalla en el Spectrum (24 filas de 32 caracteres), a cada carácter le corresponden 8 bytes del archivo, donde cada byte contiene información de una línea del carácter. Cada bit de un byte corresponde a un punto de la línea del carácter, que será negro si el bit es uno y blanco si es cero. Por ejemplo, como la forma binaria de 255 es 11111111 si hacemos "POKE 16384,255" veremos la primera línea del carácter de la esquina superior izquierda de la pantalla completamente negra. Probando otros valores se verá el efecto que producen.

La forma como están ordenados estos bytes en el archivo es un tanto extraña. Los primeros 32 bytes corresponden a las líneas superiores de los caracteres de la primera fila. Por lógica, parece que a continuación deberían venir las segundas líneas de esos mismos caracteres, luego las terceras, etc., hasta completar la primera fila.

Sin embargo, esto no es así por razones técnicas. A continuación de las primeras líneas de la primera fila de caracteres vienen las primeras líneas de la segunda fila de caracteres, luego las primeras líneas de la tercera fila de caracteres, y así hasta la octava fila.

Entonces vienen las segundas líneas de la primera fila de caracteres, luego las de la segunda fila, tercera..., hasta la octava fila. Luego las terceras líneas siguiendo el mismo esquema, las cuartas, y hasta las octavas líneas (las inferiores) de modo que se completa el tercio superior de la pantalla.

A continuación viene el segundo tercio de la pantalla con la misma estructura que el primero, y por fin el último tercio de la pantalla (el inferior).

Esta estructura es un poco complicada a primera vista. Para ayudar a comprenderla será útil entrar y correr en el Spectrum el siguiente programa BASIC:

```
10 FOR N = 16384 TO 22527
20 POKE N, 255
30 PAUSE 20
40 NEXT N
```

Lo único que hace es ir llenando por orden las sucesivas posiciones de memoria del archivo de pantalla con 255 (11111111 binario). Cambiando 255 por otro número se colocará en pantalla su forma binaria, tal como hemos visto.

Dentro de cada uno de los tercios en que se divide la pantalla, la posición de la siguiente línea de un carácter determinado se obtiene sumando a su dirección de memoria 32 × 8. Así las direcciones de las líneas del primer carácter (el de la esquina superior izquierda) son 16384, 16640, 16896, 17152, 17408, 17664, 17920 y 18176.

Para hacer un «PRINT» de un carácter en la pantalla sin tener que preocuparnos por esta extraña estructura, podemos recurrir a la rutina que lo hace para el intérprete del BASIC, que está situada en «RST $10». La posición donde se efectúa el «PRINT» viene determinada por la variable de sistema «S—POSN» de dos bytes, cuyo significado se explica en el Manual del Spectrum. Como el Spectrum puede hacer «PRINT» tanto en la parte superior como en la inferior (Inputs) de la pantalla, hay que colocar previamente al RST la variable de sistema TVFLAG a cero para efectuar «PRINT» en la mitad superior, con «XOR A» y «LD (TVFLAG),A». Los atributos (PAPER, INK, FLASH, etc.) se toman de la variable de sistema «ATTR—T».

Las características de color de PAPER, INK, FLASH, BRIGHT, están definidas en otro archivo que llamaremos de ATRIBUTOS, que ocupa desde la dirección de memoria 22528 hasta 23295 inclusive. Estos atributos están definidos en forma de uno por carácter (no de uno por punto) y su disposición en el archivo es más normal, igual que la estructura de pantalla del ZX81 con 16K pero sin marcas de fin de línea (ver Cap. 15). Cada byte define los atributos de un carácter en la forma en que se explica en el Manual del Spectrum para la función «ATTR».

El sistema del archivo de pantalla del Spectrum es más complicado que el del ZX81 pero ofrece muchísimas más posibilidades.

NOTA.— Con «RST $10» se puede poner en pantalla cualquier carácter definido por su código de la lista del apéndice A del Manual, incluso caracteres de control, colores y comandos BASIC de varias letras.

V.2. El teclado en el ZX Spectrum

El teclado está montado en el Spectrum con un sistema totalmente diferente del empleado en el ZX81.

Hay dos formas prácticas de explorar el teclado prescindiendo del BASIC en el Spectrum. La primera es aprovechando las rutinas de ROM que hacen este trabajo, de forma análoga a como se describe para el ZX81 en el capítulo 15. Aquí disponemos de tres rutinas diferentes:

KEYBOARD SCANNING: Dirección $028E. Es la que se emplea en «INKEY$». El registro E contiene después de ejecutarla un valor entre $00 y $27, diferente para cada una de las 40 teclas del teclado, o el valor $FF si no hay ninguna tecla pulsada. El registro D contiene al final un valor que indica que ninguna tecla «SHIFT» está pulsada ($FF) o cuál de ellas está siendo pulsada simultáneamente a la otra tecla. Si se pulsan las dos teclas de «SHIFT», los registros D y E contendrán los valores para «CAPS» y «SYMBOL» respectivamente.

El flag Z se pone a cero si se pulsan más de dos teclas a la vez, o si se pulsan dos teclas en las que no hay ningún «SHIFT».

KEYBOARD: Esta rutina se ejecuta automáticamente cada vez que tiene lugar una interrupción de tipo «no enmascarable», unas 50 veces por segundo (de forma parecida a como trabaja la rutina DISPLAY en el ZX81). Explora el teclado y lo decodifica, guardando el resultado en la variable de sistema «LAST—K». Cada vez que se cambia el valor de esta variable, el bit 5 de la variable de sistema «FLAGS» se pone a uno. Por tanto, para leer el teclado no hay más que consultar la variable de sistema «LAST—K».

Cuando no interesa que esta rutina se ejecute, se puede inhibir deshabilitando las interrupciones no enmascarables con la instrucción «DI», con lo que además se ganará algo de velocidad en el ZX Spectrum en CM. Para volver a habilitarla, y en todo caso para volver el control al BASIC, hay que emplear la instrucción «EI».

DECODE: Dirección $0333. Esta rutina traduce los valores que proporciona «KEYBOARD SCANNING» en el código real de la tecla pulsada, cuya lista se encuentra en el apéndice A del Manual del Spectrum. Para que funcione, los registros deben estar cargados de la siguiente forma antes de llamarla:

E: el valor obtenido en este registro en KEYBOARD SCANNING
D: el valor de la variable de sistema FLAGS
C: el valor de la variable de sistema MODE
B: el valor obtenido en el registro D tras KEYBOARD SCANNING

El código de la tecla pulsada, teniendo en cuenta todos estos parámetros, se obtiene en el registro A al final de la rutina.

Otra forma de explorar el teclado, más rápida y que permite detectar varias teclas simultáneas, es a través de la instrucción «IN», que en el Spectrum tiene un equivalente en BASIC. Las direcciones y la forma de consulta se encuentran en el Manual del Spectrum, en el capítulo dedicado a las instrucciones «IN» y «OUT». El sistema para utilizar las direcciones en CM es el siguiente:

```
LD BC, nn ... «nn» = dirección de la sección a explorar.
IN A, (C)
BIT b, A ... «b» = bit que corresponde a la tecla explorada.
```

Cuando está pulsada la tecla en cuestión, el flag Z se pone a cero. Para explorar más de una tecla sólo hay que repetir esta secuencia para cada una, de una forma parecida a como se explica para el ZX81 en el capítulo 15.

V.3. El altavoz y las conexiones «MIC» y «EAR» en el Spectrum

En el capítulo del Manual del Spectrum dedicado a las instrucciones «IN» y «OUT» se explica en qué direcciones hay que explorar o dónde dirigir los datos mediante «IN» y «OUT», para realizar la gestión de estos periféricos. En código máquina emplearemos también «IN» para explorar y «OUT» para enviar información al exterior, teniendo en cuenta que la dirección hay que cargarla previamente en el par BC y luego utilizar «IN A, (C)» ó «OUT (C), r» según el caso.

Para la gestión del altavoz se puede aprovechar la rutina de la ROM que lo hace para el BASIC. Empieza en la dirección de memoria

$03F8 y requiere que los valores de frecuencia y duración estén colocados previamente en la cima de la pila o «stack» del calculador en coma flotante (ver Capítulo 20) antes de llamar a la rutina. El dato de la cima de la pila («stack») del calculador debe ser la frecuencia y el de «debajo» la duración. Corresponde a la rutina del comando «BEEP» y, al igual que en BASIC, no se puede interrumpir ni hacer nada más mientras la nota está sonando.

V.4. Consideraciones para aplicar el contenido del libro al ZX Spectrum

En principio, todo lo explicado para el ZX81 es válido para el ZX Spectrum si no se indica lo contrario en este apéndice.

1. El mapa de memoria es evidentemente distinto para el Spectrum. Muchos de los bloques lógicos en que se divide son los mismos (aunque cambian las direcciones) y hay además otros nuevos. Está perfectamente explicado en el Manual del Spectrum.

2. Las direcciones de las subrutinas de la ROM son también distintas. Los RST coinciden casi todos con los del ZX81 pero el resto no. Las más importantes se describen en estos apéndices.

3. Afortunadamente, y como consecuencia de que en el Spectrum no hay rutina de DISPLAY, no hay registros «prohibidos», de forma que se pueden usar libremente, salvo las restricciones para el par H'L' indicadas en el Manual.

4. La localización de los programas en CM presenta muchas menos complicaciones en el Spectrum, gracias a la facilidad de guardar en cassette cualquier parte de RAM con la instrucción BASIC «LOAD...CODE» y «SAVE...CODE». Lo más práctico es reservar espacio por encima de la variable de sistema «RAMTOP» con «CLEAR nn», donde «nn» es la última dirección de memoria accesible al BASIC.

5. Algunas instrucciones del juego del Z80 se emplean de forma un poco diferente en el Spectrum:

«DI»: Deshabilita las interrupciones no enmascarables. En el Spectrum el efecto es que la rutina KEYBOARD, que explora y decodifica el teclado, no se ejecuta automáticamente cada 20 ms., con lo que ganamos un poco de velocidad extra en la ejecución del CM.

«EI»: Habilita las interrupciones no enmascarables si estaban deshabilitadas. En el Spectrum tiene el efecto de volver a permitir que

la rutina KEYBOARD se active 50 veces por segundo. Al volver al BASIC esta rutina debe estar activa, porque de lo contrario no podremos utilizar el teclado.

«HALT»: Cuando las interrupciones no enmascarables están habilitadas (con «EI») detiene todo el proceso hasta la siguiente interrupción, y entonces sigue normalmente. Es útil para sincronizar acontecimientos en momentos determinados.

«IN» y «OUT»: Sus aplicaciones especiales se explican con detalle en los apartados V.2 y V.3 de este Apéndice.

V.5. Modificaciones en el juego COMECOCOS

El principio de funcionamiento es perfectamente aplicable al Spectrum, aunque obviamente habrá que efectuar una serie de cambios.

Para poner el laberinto en el archivo de pantalla, se utilizará la instrucción «RST $10» en la forma explicada en el apartado V.1, teniendo en cuenta que aquí no hay marcas de fin de línea.

Hay que cambiar los códigos de los caracteres que representan los objetos a representar. En el Spectrum tenemos la posibilidad de utilizar caracteres definidos por el usuario, asignándolos a códigos.

Hay que relocalizar todo el programa y la situación de todas las variables internas y de sistema empleadas.

Hay que sustituir el sistema de exploración del teclado por otro adecuado al Spectrum, utilizando las explicaciones dadas en el apartado V.2.

La modificación más importante es consecuencia de que las estructuras del archivo de pantalla en el ZX81 y en el Spectrum son completamente diferentes, y el programa, en su versión para el ZX81, explora directamente ese archivo. Un sistema para adaptar el juego puede ser construir en alguna parte una especie de «archivo de pantalla» con las características del diseñado para el ZX81 (marcas de fin de línea incluidas), donde el programa explorará y moverá los objetos. Entonces se montará una rutina a base de «RST $10» que copie este «archivo fantasma» en el verdadero archivo de pantalla del Spectrum (excluyendo las marcas de fin de línea) y se efectuará una llamada a esta nueva subrutina dentro del LAZO PRINCIPAL DE DESARROLLO y en las otras ocasiones en que cambie algo en la pantalla (por ejemplo, en algunas partes del CONTROL DE FINALES).

Dejo estas modificaciones como un ejercicio que dará ocasión para revisar y comprobar que se han entendido muchos de los conceptos explicados en el libro.

OBRAS DE EDICIONES TECNICAS REDE

ACUSTICA Y BAJA FRECUENCIA

ALTA FIDELIDAD A BAJO COSTE (Libro n.° 87)
212 págs., 117 figs. Redacción REDE.

AUDIO REPARACION (Libro n.° 126)
170 págs., 187 figs. Autor: F. Mor.

COMUNICACION INSTANTANEA (Libro n.° 109)
114 págs., 54 figs. Redacción REDE.

CONSTRUCCION PRACTICA Y SIMPLIFICADA DE CAJAS ACUSTICAS PARA HI-FI (Libro n.° 138)
122 págs., 79 figs. Redacción REDE.

INICIACION AL DISEÑO DE CIRCUITOS DE AUDIO (Libro n.° 148)
108 págs., 42 figuras. Redacción REDE.

MAGNETOFONOS Y CASSETTES (Libro n.° 151)
168 págs., 75 figuras. Redacción REDE.

MUSICA ELECTRONICA (Libro n.° 83)
150 págs., 70 figs. Redacción REDE.

CIRCUITOS COMPROBADOS

Un tipo de libro sin precedentes. Cada montaje, antes de ser descrito, ha sido realizado y mantenido en óptimo funcionamiento en el Laboratorio de Experimentación de REDE. Tamaño de cada volumen: 210 × 270 mm.

AUDIO-1 (Libro n.° 111)
84 págs., 100 figs. Redacción REDE.

MONTAJES PRACTICOS (Libro n.° 115)
84 págs., 121 figs. Redacción REDE.

PRACTICA DIGITAL (Libro n.° 118)
80 págs., 134 figs. Redacción REDE.

CIRCUITOS INTEGRADOS (Libro n.° 128)
84 págs. 120 figs. Redacción REDE.

AUDIO-2 (Libro n.° 131)
88 págs., 100 figs. Redacción REDE.

JUEGOS ELECTRONICOS-I (Libro n.° 132)
80 págs. 100 figs. Redacción REDE.

ANTI-ROBO (Libro n.° 135)
76 págs., 114 figs. Redacción REDE.

JUEGOS ELECTRÓNICOS-II (Libro n.° 141)
80 págs., 96 figs. Redacción REDE.

AUDIO-3 (Libro n.° 143)
82 págs., 116 figs. Redacción REDE.

TELEMANDO (Libro n.° 146)
80 págs. 112 figs. Redacción REDE.

COMPROBADORES (Libro n.° 152)
84 págs. 100 figuras. Redacción REDE.

AUDIO-4 (Libro n.° 154)
80 págs., 100 figs. Redacción REDE.

ELECTRONICA EN EL AUTOMOVIL (Libro n.° 158)
84 pág., 96 figs. Redacción REDE.

LUCES SICODELICAS Y JUEGOS LUMINOSOS
(Libro n.° 160)
82 págs., 100 figs. Redacción REDE.

COMPONENTES ELECTRONICOS

EL TIRISTOR: APLICACIONES, CARACTERISTICAS Y FUNCIONAMIENTO
(Libro n.° 73)
104 págs., 50 figs. Autor: R. Swoboda.

LA FIABILIDAD DE LOS COMPONENTES ELECTRÓNICOS (Libro n.° 70)
162 págs., 28 figs., 31 tablas. Autor: C. E. Jowet.

ELECTRICIDAD Y MEDICION

INSTALACIONES ELECTRICAS DE BAJA TENSION
(Libro n.° 64)
136 págs. Autores: A. Bandini y M. Bertolini.

LUMINOTECNIA (Libro n.° 58)
177 págs., 87 figs., 46 tablas. Autor: G. Clerici.

MANTENIMIENTO DE EQUIPOS ELECTRICOS
(Libro n.° 14)
111 págs., 82 figs., 20 tablas. Autor: P. L. Cerato.

MEDIDAS ELECTRICAS (I/I):
METODOS E INSTRUMENTOS (Libro n.° 57)
328 págs., 194 figs. 50 tablas. Autor: A. Bandini.

MEDICION ELECTRICA (I/II):
METODOS E INSTRUMENTOS (Libro n.° 62)
288 págs., 153 figs. 42 tablas. Autor: A. Bandini.

MEDICION ELECTRICA (II):
ENSAYOS DE MAQUINAS (Libro n.° 59)
400 págs., 120 figs. Autores: A. Bandini y M. Bertolini.

METODOS DE MEDIDA EN CIRCUITOS DE CORRIENTE CONTINUA (Libro n.° 54)
139 págs., 90 figs. Autor: A. Bossi.

SEÑALIZACIONES ELECTRICAS
(Libro n.° 32) 226 págs., 132 figs. Autor: G. Clerici.

ESQUEMARIOS

Cada volumen constituye un elemento insustituible tanto para el reparador de TV b/n y color como para los especializados en magnetófonos y cassettes, incluyendo esquemas y notas de servicio.

TV Blanco y negro

ESQUEMARIO TV/I	(libro n.° 21)
ESQUEMARIO TV/II	(libro n.° 22)
ESQUEMARIO TV/III	(libro n.° 55)
ESQUEMARIO TV/IV	(libro n.° 67)
ESQUEMARIO TV/V	(libro n.° 76)
ESQUEMARIO TV/VI	(libro n.° 81)
ESQUEMARIO TV/VII	(libro n.° 88)
ESQUEMARIO TV/VIII	(libro n.° 94)
ESQUEMARIO TV/IX	(libro n.° 100)
ESQUEMARIO TV/X	(libro n.° 105)
ESQUEMARIO TV/XI	(libro n.° 113)
ESQUEMARIO TV/XII	(libro n.° 127)
ESQUEMARIO TV/XIII	(libro n.° 140)
ESQUEMARIO TV/XIV	(libro n.° 155)

TV Color

ESQUEMARIO TVC/I	(libro n.° 112)
ESQUEMARIO TVC/II	(libro n.° 114)
ESQUEMARIO TVC/III	(libro n.° 116)
ESQUEMARIO TVC/IV	(libro n.° 117)
ESQUEMARIO TVC/V	(libro n.° 119)
ESQUEMARIO TVC/VI	(libro n.° 129)
ESQUEMARIO TVC/VII	(libro n.° 134)
ESQUEMARIO TVC/VIII	(libro n.° 136)
ESQUEMARIO TVC/IX	(libro n.° 137)
ESQUEMARIO TVC/X	(libro n.° 142)
ESQUEMARIO TVC/XI	(libro n.° 145)
ESQUEMARIO TVC/XII	(libro n.° 150)
ESQUEMARIO TVC/XIII	(libro n.° 157)
ESQUEMARIO TVC/XIV	(libro n.° 161)
ESQUEMARIO TVC/XV	(libro n.° 162)
ESQUEMARIO TVC/XVI	(libro n.° 165)

Magnetófonos y Cassettes

ESQUEMARIO I	(libro n.° 85)	ESQUEMARIO V	(libro n.° 133)
ESQUEMARIO II	(libro n.° 103)	ESQUEMARIO VI	(libro n.° 139)
ESQUEMARIO III	(libro n.° 123)	ESQUEMARIO VII	(libro n.° 144)
ESQUEMARIO IV	(libro n.° 130)	ESQUEMARIO VIII	(libro n.° 159)

LIBROS-HERRAMIENTA

Obras destinadas al reparador de TV, radio, auto-radio, electrodomésticos, magnetófonos, etc., al montador y al experimentador.

ALARMA ELECTRONICA (libro n.° 102)
150 págs. 91 figs. Redacción REDE.

AUTOMATISMOS DE FACIL CONSTRUCCION (Libro n.° 107)
128 págs. 89 figs. Redacción REDE.

BIOELECTRONICA (Libro n.° 149)
132 págs., 72 figs. Redacción REDE.

COMODIDADES ELECTRONICAS DE FACIL MONTAJE (Libro n.° 99)
124 págs. 75 figs. Redacción REDE.

CONTRAESPIONAJE ELECTRONICO (Libro n.° 93)
115 págs., 50 figs. Redacción REDE.

ELECTRONICA EN LA FOTOGRAFIA (libro n.° 121)
126 págs., 52 figs. Redacción REDE.

ELECTRONICA AL SERVICIO DEL AUTOMOVILISTA, LA (Libro n.° 122)
146 págs. 68 figs. Redacción REDE.

ESPIONAJE ELECTRONICO (Libro n.° 84)
128 págs. 62 figs. Redacción REDE.

IMPROVISACIONES QUE DAN DINERO Y AHORRAN TIEMPO
Volumen I (libro n.° 48): 140 págs., 91 figs.
Volumen II (libro n.° 63): 126 págs., 77 figs.
Volumen III (libro n.° 80): 130 págs., 82 figs.
Volumen IV (libro n.° 98): 146 págs., 90 figs.
Volumen V (libro n.° 104): 138 págs., 91 figs.

JUGUETES ELECTRÓNICOS - I (Libro n.° 96)
122 págs., 61 fig. Redacción REDE.

JUGUETES ELECTRONICOS - II (Libro n.° 106)
128 págs. 91 figs. Redacción REDE.

PRACTICA ELECTRONICA SIMPLIFICADA Y EXPERIMENTAL: CON 1 TRANSISTOR, MULTIPLES MONTAJES COMPROBADOS (Libro n.° 120)
118 págs. 62 figs. Redacción REDE.

PRACTICA ELECTRONICA SIMPLIFICADA Y EXPERIMENTAL: CON 2 TRANSISTORES, MULTIPLES MONTAJES COMPROBADOS (Libro n.° 124)
140 págs. 67 figs. Redacción REDE.

PRACTICA ELECTRONICA SIMPLIFICADA Y EXPERIMENTAL: CON 3 TRANSISTORES, MÚLTIPLES MONTAJES COMPROBADOS (Libro n.° 125)
132 págs. 64 figs. Redacción REDE.

RECUPERACIÓN DE COMPONENTES ELECTRONICOS (Libro n.° 156)
166 págs., 91 figs. Redacción REDE.

REPARACION DE ELECTRODOMESTICOS (Libro n.° 40)
265 págs., 167 figs. Autor: E. Tricomi.

SOLDADURA ELECTRICA EN LOS MONTAJES ELECTRONICOS (Libro n.° 147)
104 págs., 69 figs. Autor: F. Mor.

SEGURIDAD ELECTRONICA (Libro n.° 108)
120 págs. 60 figs. Redacción rede.

UHF, TECNICA/ADAPTACION/REPARACION (Libro n.° 33)
335 págs. 302 figs. Autor: F. Möhring.

TELEVISION

PRACTICA DE LA CONSTRUCCION E INSTALACION DE ANTENAS DE FM Y DE TV (Libro n.° 92)
272 págs. 222 figs. Redacción REDE.

REPARACION TV-I (Libro n.° 77)
300 págs. 472 figs. Autor: F. Mor.

REPARACION TV-II (Libro n.° 110)
304 págs. 470 figs. Autor: F. Mor.

REPARACION TVC (Libro n.° 153)
140 págs. Ilustraciones a todo color. Redacción REDE.

TELEVISION COLOR BASICA (Libro n.° 166)
170 págs., 80 figs. Ilustraciones a todo color.
Autor: B. Miguel.

VARIOS

CONTROL NUMERICO DE LAS MAQUINAS HERRAMIENTAS (Libro n.° 86)
90 págs., 64 figs. Autor: M. Flego.

DIBUJO INDUSTRIAL (Libro n.° 35)
260 págs., 158 figs., 56 tablas. C. Clerici.

DIBUJO TECNICO (Libro n.° 15)
154 págs., 82 figs., 20 tablas. C. Clerici.